

# Process Algebra

## A Petri-Net-Oriented Tutorial

Eike Best<sup>1</sup> and Maciej Koutny<sup>2</sup>

<sup>1</sup> Parallel Systems, Faculty of Computing Science  
Carl von Ossietzky Universität Oldenburg  
D-26111 Oldenburg, Germany

[Eike.Best@informatik.uni-oldenburg.de](mailto:Eike.Best@informatik.uni-oldenburg.de)

<sup>2</sup> School of Computing Science, University of Newcastle  
Newcastle upon Tyne NE1 7RU, United Kingdom  
[Maciej.Koutny@newcastle.ac.uk](mailto:Maciej.Koutny@newcastle.ac.uk)

**Abstract.** Process algebras aim at defining algebraic calculi for concurrency and communication between concurrent processes. This paper describes some of the issues that would seem to be worth discussing when process algebraic ideas are related to Petri net theoretical concepts.

**Keywords:** Petri nets, process algebras.

## 1 Introductory Remarks

Many research monographs and textbooks have been written on the subject of *process algebras* (see, for instance, [3, 6, 19, 20, 24–26]), and even a comprehensive handbook of 1342 pages was published three years ago [4]. Thus, even when the topic is restricted to the relationship between process algebras and Petri nets, there is a vast amount of work that cannot be described in short a tutorial. We therefore chose to write an account of *selected* thoughts that might occur to someone attempting to form a link between process algebras and Petri nets; moreover, while aiming at an easy-to-be-followed presentation, we included several pointers to the relevant papers and results. We will discuss a well-known concrete process algebra (the reader should be aware that there are several of them, just as there are several classes of Petri nets), and highlight some of the key issues both in the design of such an algebra, and in its translation into Petri nets.

Process algebras have been studied for about 25 years, but some ideas behind them go back even further; indeed, even regular expressions can be viewed as a very simple process algebra equivalent to finite state automata. First seeds for what has by now become a standard approach to process algebra have been sown by Robin Milner [24], based on his work on flow graphs, and Tony Hoare [18, 19], based on his ideas about designing concurrent programs. Also, at around the same time, the relationship between the path notation, which originates from operating systems design (Roy Campbell and Nico Habermann [8]), and Petri nets has been pioneered by Peter Lauer and others [20]. Soon after that, research groups in Amsterdam started to investigate an axiomatic approach to process algebra [3].

We here concentrate on the relationship between process algebras and Petri nets which, in particular, could allow one to apply Petri-net-based analysis methods (such as the S-invariant method) in the process algebraic framework, and to transfer concepts of concurrency and causality that are well-developed in Petri net theory into the domain of process algebras. Apart from the already mentioned work on path expressions, some early influential research in this area has been carried out by Ursula Goltz [17], Gerard Boudol and Ilaria Castellani [7], Rob van Glabbeek and Frits Vaandrager [16], Ugo Montanari and his research group in Pisa [11], and Ernst-Rüdiger Olderog [26]. Much of this work aimed at giving a faithful translations for the existing process algebras, the majority of which had been designed without Petri nets in mind. By contrast, but also building upon this earlier work, the ‘box algebra’ approach of [6] on which this paper is based, aimed at designing a ‘Petri nets-friendly’ process algebra.

## 2 Basic CCS

As a basis for our discussion, we have chosen the original *Calculus of Communicating Systems* by Robin Milner [24, 25], referred to as the (*basic*) *CCS*. The word ‘basic’ here reflects the fact that there exist numerous extensions (e.g., [2, 9]) as well as interesting restrictions (e.g., [10]) of CCS and related process algebras. CCS is *action-based*, which means that it is built upon a set of (*atomic*) ‘observable’ actions  $A = \{a, \hat{a}, b, \hat{b}, c, \hat{c}, \dots\}$  together with a distinct ‘unobservable’ (or ‘silent’) action  $\tau$ . Every observable action  $a$  is coupled with another observable action, denoted by  $\hat{a}$  and called its *conjugate* (and  $\hat{a}$  has  $a$  as its conjugate, i.e.,  $a = \hat{\hat{a}}$ ). (Though Milner’s CCS uses over-barring, such as  $\bar{a}$ , to denote conjugates, we changed this notation to over-hatting in order to avoid potential confusion with another over-barring, to be introduced later on.)

CCS provides a method for structuring a collection of actions in such a way that some overall coordinated behaviour is described in order, e.g., to specify the desired behaviour of some still-to-be-designed system, or to describe crucial aspects of the behaviour of some existing system. Technically, this is achieved by imposing a syntax on top of the set of actions, as follows:

$$E ::= \text{nil} \mid E + E \mid z.E \mid E \mid E \mid E \backslash a .$$

In the above,  $a$  ranges over  $A$ , and  $z$  over  $A \cup \{\tau\}$ . The syntax defines CCS *process expressions* (or just *processes*)  $E$ , and it should be read in the usual Backus-Naur style. Thus, for example,  $\text{nil}$  is a process, and if  $E$  and  $F$  are processes then so is  $E + F$ . Within the syntax, symbols ‘+’ and ‘|’, as well as constructs ‘ $z.$ ’ and ‘ $\backslash a$ ’ denote *process algebraic operators*.

There is a single basic process  $\text{nil}$  without any operator at all which can be viewed, in the usual way, as a nullary operator, or a *constant*. There is, furthermore, an infinite family of unary operators, parameterised by actions (the *prefixing operators*,  $z.$ ). Another family of unary operators, written in postfix style, are the *restrictions*  $\backslash a$  (one for each  $a \in A$ ). The only binary operators, written in infix notation, are the *choice* ‘+’ and *composition* ‘|’. To avoid an excessive

use of the parentheses, it is assumed that prefixing binds more tightly than choice, and that choice binds more tightly than composition; thus, for instance,  $a.b.\text{nil} + c.\text{nil} \mid d.\text{nil}$  stands for  $((a.(b.\text{nil})) + (c.\text{nil})) \mid (d.\text{nil})$ .

*Process variables*, such as  $X$ , are also considered to be basic processes, each such variable being associated with a unique defining expression of the form  $X \stackrel{\text{df}}{=} E$ . Variables provide support for different levels of abstraction within a CCS specification, as well as for recursion: e.g.,  $X$  with  $X \stackrel{\text{df}}{=} a.X$  is a recursive expression.

Every process has an associated set of behaviours. It is one of the main objectives of *formal semantics* of a process algebra to make this notion precise. The behaviour of process expressions is often given in terms of ‘can make a move’ statements, which are similar to ‘an action can be executed’ statements in general concurrent systems terminology, or ‘a transition can occur’ statements in Petri net terminology (for example, the process  $a.\text{nil}$  can make move  $a$ , and thereafter no further move). Referring to CCS syntactic constructs, we have that the basic process  $\text{nil}$  can make no move at all;  $E + F$  behaves either like  $E$  or like  $F$ ;  $z.E$  can make the move  $z$  and then behaves like  $E$ ;  $E \mid F$  behaves like both  $E$  and  $F$ , together with some further (synchronised) activities which will be described later;  $E \backslash a$  behaves like  $E$ , except that the actions  $a$  and  $\hat{a}$  are blocked; and, finally,  $X$  behaves like the process  $E$  in the equation  $X \stackrel{\text{df}}{=} E$ .

A straightforward attempt to capture the behaviour associated with a process expression could be to emulate the approach taken when the language of a regular expression is defined. However, this might lead to identifying processes which may behave rather differently in certain contexts (see the discussion in Section 4). To address this problem, CCS uses more refined scheme for describing the possible moves of a process, namely that offered by structural operational semantics (SOS), pioneered by Gordon Plotkin [28]. In SOS, the concept of *inference rules* of formal logic is applied to processes, using the fact that if a process  $E$  makes a move, then the ‘remaining’ behaviour of  $E$  after making this move can be described by another process expression  $E'$ . For example,  $E = a.\text{nil} + b.c.\text{nil}$  can make move  $b$ , and the remaining behaviour can be described by process  $E' = c.\text{nil}$  (note that the left-hand side branch of the  $+$  together with the  $+$  itself, has disappeared because it was not selected, and that the  $b$  has disappeared because it was executed). In general, we say that ‘ $E$  makes a move  $z$  and becomes  $E'$ ’, and write this formally as  $E \xrightarrow{z} E'$ . For processes, then, the SOS rules have the following format: *if* some expression  $E$  can make a move and become  $E'$  (the rule’s *premise*), *then* some expression  $D$  (syntactically related to  $E$ ) can also make a move and become  $D'$  (the rule’s *conclusion*). A rule is divided by a horizontal bar, above which its premise, and below which its conclusion are given. For example, the two rules for choice are:

$$\frac{E \xrightarrow{z} E'}{E + F \xrightarrow{z} E'} \quad \text{and} \quad \frac{F \xrightarrow{z} F'}{E + F \xrightarrow{z} F'}$$

In this vein, one may go through the syntax providing suitable rule(s) for each syntactic clause, i.e., for each operator (or family of operators). For example, in case of restriction we have a single rule:

$$\frac{E \xrightarrow{z} E', z \neq a, z \neq \hat{a}}{E \setminus a \xrightarrow{z} E' \setminus a}$$

Thus  $E \setminus a$  can make a  $z$ -move if  $E$  can and  $z$  is different from  $a$  and  $\hat{a}$ . It is also worth noting that there are no rules at all for the process  $\text{nil}$ .

### 3 From Process Expressions to Petri Nets

We now will go through the basic CCS process algebra and discuss several of the concepts contained in it, as well as some of the issues pertaining to its connection to Petri nets. The discussion of infinite behaviour is deferred to the last section of this paper.

**Sequential Composition.** The basic CCS means of generating sequential behaviour is through the prefixing operator. For instance,  $a.b.\text{nil}$  can ‘make an  $a$ -move’, thereafter ‘make a  $b$ -move’, and then terminate:

$$a.b.\text{nil} \xrightarrow{a} b.\text{nil} \xrightarrow{b} \text{nil}.$$

It may be observed here that  $a.b.\text{nil}$ ,  $b.\text{nil}$  and  $\text{nil}$  are expressions with different *structure*, indicating that a CCS expression may be subjected to a structural change through behaviour (in general, the structure and the behaviour of CCS expressions are closely intertwined). Moreover, it is fairly clear that a Petri net equivalent of  $a.b.\text{nil}$  should be something like the transition-labelled net shown on the left-hand side of Figure 1. Note that we do need the labelling as one might write a process such as  $a.a.\text{nil}$ , giving rise to two different transitions with the same label  $a$ .



**Fig. 1.** Net for  $a.b.\text{nil}$  before and after the occurrence of  $a$ .

There is a conceptual discrepancy in thinking about structure and behaviour in CCS and in Petri nets. After  $a$  has occurred in the net, the transition labelled  $a$  is still visible in its structure (see the right-hand side of Figure 1), though it cannot be activated anymore, whilst the expression  $a.b.\text{nil}$  changes to  $b.\text{nil}$  after the occurrence of  $a$ , and thus the  $a$  is ‘lost’ (this way of viewing behaviour is

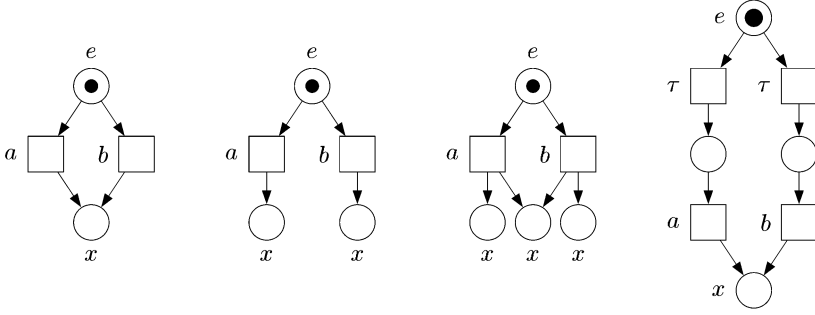
actually quite particular to the standard SOS semantics rather than to process algebra as such).

What would happen if we wished to put more complicated partial expressions into sequence, rather than just two actions  $a$  and  $b$ ? The CCS syntax tells us that having  $a$  in sequence with something arbitrarily complicated is possible. However, it explicitly disallows anything more complicated than just an action to be written *in front* of the prefix operator symbol. For instance, writing  $(a + b).nil$  is forbidden. Theoretically speaking, this is not too restrictive, because the general sequential composition can be ‘emulated’, using in an essential way the still-to-be-explained composition operator and restriction. Nevertheless, it may be useful to have a more general sequential operation at one’s disposal, and there are some process algebras with such an operator as a basic feature, e.g., [3].

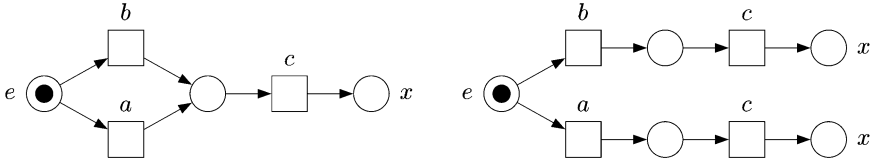
**Choice Composition.** This seems to be a rather innocent operation. However, as we will see, some care needs to be taken when it is applied in combination with other constructs. Let us first see how a simple choice between  $a$  and  $b$ , such as  $a.nil + b.nil$ , can be expressed in terms of Petri nets. We haven’t yet considered the  $nil$  by itself, but after looking at its role in the previous example, it should indicate ‘the end of a net (or of net’s execution)’. We emphasised this by naming the place to which  $nil$  corresponds  $x$  (for ‘exit’, as opposed to the place where the initial token resides, which is called  $e$ , for ‘entry’).

Given  $a.nil + b.nil$ , should the corresponding net have only one end, or two, or even three ends? All three possibilities are shown in Figure 2, and all of them can be found in the literature. Moreover, when the choice is embedded in a context, we might wish to model it as being an *internal* one, by which is meant that the process performing the choice should be free to ‘make up its mind’ without being hindered by any external influence. A good way of modelling internal choice is shown on the right-hand side of Figure 2. The choice between  $a$  and  $b$  is ‘pre-poned’ there, in the sense that it depends on a prior choice between two silent  $\tau$  actions (there are process algebras in which one may distinguish syntactically between external and internal choice, such as TCSP [19]). However, such a translation is inappropriate for CCS: Indeed, CCS has another expression corresponding to this net,  $\tau.a.nil + \tau.b.nil$ , which needs to be distinguished carefully (although we have not yet explained exactly why) from the original expression  $a.nil + b.nil$ .

Now, which of the three possibilities (a/b/c) in Figure 2 is to be preferred as a translation of  $a.nil + b.nil$ ? Cases can be made for all of them: Figure 2(b) [11] seems to be the closest Petri net correspondent of that expression, since there are two  $nil$ ’s in it and two  $x$  places in the net; Figure 2(c) has strong category theoretical (if not categorical) arguments in its favour [33]; and Figure 2(a) [6] treats choice as a forward/backward symmetric sequential construct (almost as a nondeterministic **if-fi** statement in programming [12], or like  $a + b$  in a regular expression). Moreover, the latter allows the net to be used as a building block in the following sense. An expression such as  $(a + b);c$  where the semicolon denotes sequential composition, would give, using Figure 2(a), the left-hand side



**Fig. 2.** Nets for  $(a.nil) + (b.nil)$ : (a) one-ended, (b) two-ended, (c) three-ended, and (d) with internal choice.



**Fig. 3.** Nets for  $(a + b); c$  constructed from Figure 2(a) and Figure 2(b).

of Figure 3. Figure 2(b) would not quite as easily fit such an extension; the net shown on the right-hand side of Figure 3 is close to  $(a + b); c$  and has two  $x$  places, but also contains a duplication of the  $c$  action which is not present in the process expression.

**Parallel Composition.** Putting processes in parallel for performance reasons, or for architectural reasons, has long been a challenge for theoreticians, as well as for practitioners, because the behaviour of such compound processes seems to be much more difficult to describe and to master than sequential behaviour. For instance, if two parallel processes access common variables, one needs to be able to control these accesses and their orderly interaction. Numerous special programming constructs have been invented in order to manage this, and other kinds of, interaction between parallel processes. One that plays a particularly important role in process algebra is the concept of a *handshake communication*. This is a form of a symmetric synchronisation, by which an action that is shared between, say,  $n$  processes can be executed only if *all  $n$  of them* are ready to do so. This is much like in Petri net theory, where a transition with  $n$  input places can occur only if all of these places carry a token.

Basic CCS has a particular incarnation of the handshake, in that it is always two-partnered (i.e.,  $n=2$ ), and it always results in an internal action. Moreover, handshakes can occur only between  $a$  actions and their conjugates,  $\hat{a}$  actions. The idea is, quite literally, that two conjugates fit together like two hands about

to engage in a handshake, and that the resulting  $\tau$  synchronisation describes the handshake itself. Let us have a look at the three SOS rules for CCS composition:

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F} \quad \frac{F \xrightarrow{a} F'}{E \mid F \xrightarrow{a} E \mid F'} \quad \frac{E \xrightarrow{a} E', F \xrightarrow{\hat{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

First, note carefully the difference between the first two rules and the rules for choice composition: the processes ‘remaining’ after move  $a$  still contain parts of the original processes which had not been involved in the move, while in the rule for choice these have disappeared. This models parallel (rather than nondeterministically selected) behaviour. Second, note the appealing simplicity of the third, handshake rule: if  $E$  can make an  $a$ -move and  $F$  can make an  $\hat{a}$ -move, then their composition  $E \mid F$  can make a  $\tau$ -move, creating as a new process the composition of whatever remains from  $E$  and  $F$  after making their respective moves.

For instance, what are the sequences of moves, or in technical terms, the *interleavings*, that the process  $a.a.\text{nil} \mid \hat{a}.\hat{a}.\text{nil}$  could make? If we applied only the first two rules of  $\mid$ , we would have move sequences  $aa\hat{a}\hat{a}$ ,  $\hat{a}\hat{a}a\hat{a}$ ,  $\hat{a}\hat{a}a\hat{a}$ ,  $\hat{a}\hat{a}a\hat{a}$  (and all their prefixes as well). But if we now also take the third rule into account, there are still more behaviours of that expression, such as  $\tau a\hat{a}$ . What happened here is that we let the first  $a$  on the left-hand side of  $a.a.\text{nil} \mid \hat{a}.\hat{a}.\text{nil}$  be synchronised with the first  $\hat{a}$  on the right-hand side, and the rest got executed in an unsynchronised way. Of course,  $\tau\hat{a}a$  is also a possible behaviour, and so are  $\hat{a}\hat{a}\tau$ ,  $\hat{a}a\tau$  and  $\tau\tau$ . Still more behaviour is possible if we let the left-hand side process execute its first  $a$  and then synchronise its second  $a$  with the first  $\hat{a}$  on the right-hand side, thus:  $a\tau\hat{a}$ . And, by symmetry, we may have  $\hat{a}\tau a$ . What we just witnessed suggests that a parallel composition can give rise to a bewildering amount of behaviour. In fact, using only prefixing and composition, it is possible to construct processes that give rise to exponentially many interleavings (in terms of the size of an expression). What is so bewildering about this (or perhaps, what so bewildered early researchers on the subject) is not the sheer size of the set of potential interleavings, but the wide variety of possible interferences of one process by another one at unpredictable points in time, some of which may be quite unwanted in real applications. In practice, one is interested in such a global point of view, though, in order to prove assertions about the possible set of *all* behaviours. Of course, the  $\mid$  rule does not really help here, being (as all SOS rules are, and also as the transition rule in Petri nets is) local in nature. However, such a rule is a solid formal foundation for further applications of more advanced and global methods, just as the transition rule in Petri nets is necessary for, e.g., the state equation, and the latter, for instance, for the structural analysis of nets.

Other process algebras have different parallel operators and different action synchronisation schemes. COSY and, more particularly, the path notation [8, 20], have top-level parallelism, which means that one may express a collection of parallel processes and paths, inside which the parallel operator may not occur;

i.e., no nesting of parallel composition is allowed. Synchronisation occurs through *equality of action names*, that is, if an action  $a$  occurs in two or more different processes (or, to be precise, paths), it may be executed only if all of them are ready to do so. TCSP [19] also has synchronisation on common action names – with some subtle differences to COSY – but, on the other hand, allows nested parallelism. Hoare’s original CSP [18] can be characterised as a mixture of top-level parallelism and conjugation-based two-way synchronisation. Conjugation is syntactically expressed in CSP by  $!$ -actions (send actions) and  $?$ -actions (receive actions); see [5] for an account of its translation into Petri nets. The PBC [6] generalises conjugation to  $n$ -way synchronisation, where  $n \geq 2$ .

Let us now turn to the translation of processes using the composition operator into Petri nets. Actually, it is not very hard to define such a translation. Given  $E \mid F$ , one first draws separately the nets for  $E$  and  $F$  and lays them side by side (or, in technical terms, forms their *disjoint union*). In a second step, one collects all pairs of transitions that are labelled by  $a$  in one of the components and by  $\hat{a}$  in the other component and inserts a common transition, labelled by  $\tau$ , that acts in the net of each component like the transition it originates from does. We need to do this not just for  $a, \hat{a}$ -pairs, but also for all other pairs of conjugates. For the  $a.a.\text{nil} \mid \hat{a}.\hat{a}.\text{nil}$  example, we actually have *four* matching  $a, \hat{a}$ -pairs, because there are two  $a$ ’s on the left-hand side and two  $\hat{a}$ ’s on the right-hand side, and any two of them can be matched. The construction thus yields the labelled Petri net shown in Figure 4. Virtually all translations of CCS into Petri nets that exist in the literature will yield this net; there is not really any alternative way of the sort we discussed in the case of the choice operator.

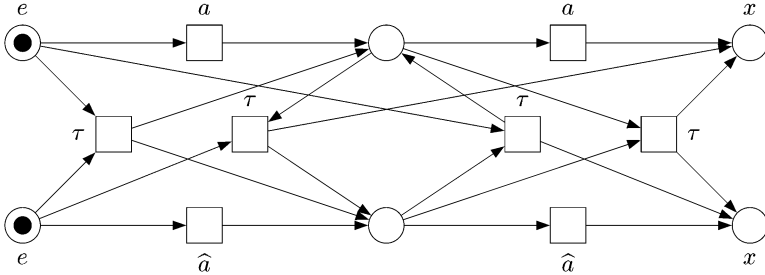


Fig. 4. Net for  $a.a.\text{nil} \mid \hat{a}.\hat{a}.\text{nil}$ .

**Restriction.** To analyse its effect, consider the process  $(b.a.\text{nil}) \backslash a$ . It can make a  $b$ -move, and then no further move, because the  $a$  has been restricted away. Note carefully that the same is true for the process  $b.\text{nil}$ , but that there is still an important difference between these two processes: the latter ‘terminates properly’, while the former ‘gets stuck’ (or, more technically, it *deadlocks* without reaching a proper final state). This difference is not visible in terms of the behavioural sequences of two terms, unless we add some information about termination.

The Petri net of a restricted process  $E \backslash a$  is simply obtained by omitting from the net of  $E$  all  $a$ -labelled and all  $\hat{a}$ -labelled transitions together with the adjacent

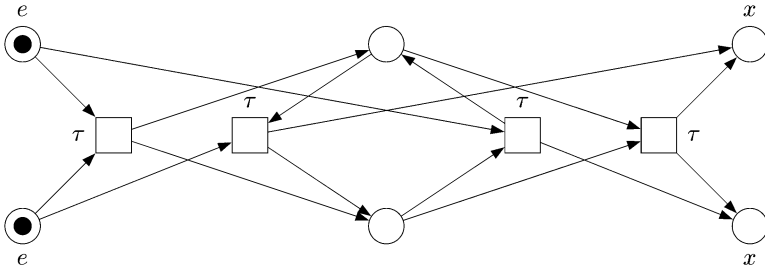


arcs. Thus, the net corresponding to  $(b.a.nil)\backslash a$  looks like that on the left-hand side of Figure 5. On the right-hand side of Figure 5, the net corresponding to  $b.nil$  is shown. The isolated  $x$  place on the left-hand side may appear superfluous, but it (and the fact that place  $s$  is *not* an exit place) captures the fact that this net cannot terminate properly, and thus describes the main distinction between processes  $(b.(a.nil))\backslash a$  and  $b.nil$ . It is therefore good to keep that place around, or at least keep the information somewhere that place  $s$  is not an exit place.



**Fig. 5.** Nets for  $(b.a.nil)\backslash a$  and  $b.nil$ .

Restriction is typically used in combination with composition **I**, the main idea being that a pair of conjugated actions is used in order to create a desired synchronisation, after which the pair is no longer needed and can be restricted away. For instance, we may consider the process  $((a.(a.nil))\mathbf{I}(\hat{a}.\hat{a}.nil))\backslash a$ , coming from the expression discussed in the last section. Restricting away the  $a$ 's (and  $\hat{a}$ 's) implies that only the  $\tau$ -behaviour of that expression still remains possible; that is, it can make two  $\tau$  moves, and that is all. Let us have a look at the corresponding net in Figure 6 to check whether this is correctly reflected. Only the first  $\tau$ -transition can occur, and then the fourth, putting both tokens on their respective  $x$  places (and thus terminating execution properly). Rightfully, the second and third  $\tau$ -transitions have become ‘dead’ (in the sense that they cannot be executed anymore), since they were both associated with a prior execution of  $a$  or, respectively, of  $\hat{a}$ , and both executions are no longer possible. One might be tempted to delete them from the net. However, such an optimisation is in general very hard to detect, and it is thus better not to incorporate it in any basic constructions, but to add it, if desired, as an appendix to the systematic construction.



**Fig. 6.** Net for  $(a.a.nil\mathbf{I}\hat{a}.\hat{a}.nil)\backslash a$ .

A similar argument as for the difference between  $(b.a.nil)\backslash a$  and  $b.nil$  can be used to justify why processes  $\tau.a.nil + \tau.b.nil$  and  $a.nil + b.nil$  should not be treated as equivalent. For suppose that either is put into the context  $(\bullet\backslash a)$ , that is, suppose that the ‘hole’ in that expression (the bullet, ‘ $\bullet$ ’) is first replaced by  $\tau.a.nil + \tau.b.nil$  and then by  $a.nil + b.nil$ . We obtain, respectively:

$$(\tau.a.nil + \tau.b.nil)\backslash a \quad \text{and} \quad (a.nil + b.nil)\backslash a .$$

The internal choice in the first of these expressions can be resolved in such a way that the  $\tau$  just in front of the  $a$  is selected, and after that, the entire expression can make no further move and is ‘deadlocked’, similarly to the expression  $(b.(a.nil))\backslash a$  after executing  $b$ . By contrast, in the second expression, no  $\tau$  can be selected, and the  $a$  cannot be selected either, because it is restricted away; thus, the  $b$  remains executable, and after executing it, a final state has been reached, so that no such deadlock is possible. Thus, the two expressions behave differently in the same context, and one should, as a rule, be careful to distinguish them. Any formal semantics that does *not* distinguish two expressions which behave differently in a given (desired) context is technically called *not fully abstract*. More precisely, if a formal semantics distinguishes two expressions whenever they behave differently in at least one context out of a predefined set of contexts (and equates them if such a context cannot be found), then it is called *fully abstract* (with respect to that set of contexts).

## 4 Algebraic Laws, and Bisimilarity

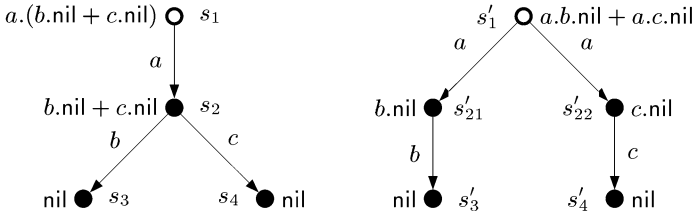
Looking at the two SOS rules for the choice operator, it is clear that processes like  $E + F$  and  $F + E$  should be semantically the same. Now, if we looked at their nets, such a conclusion would be easy to defend as the nets are just isomorphic. However, no such notion of isomorphism suggests itself easily for expressions. Here is another case:  $E\backslash a$  and  $(E\backslash a)\backslash a$ . Clearly, restricting  $a$  two times in a row will give the same result as restricting  $a$  just once; technically speaking, restriction is *idempotent*. Yet the two expressions are syntactically quite different. But if we consider their nets, then it is easy to see that they are again isomorphic, since the outer restriction in  $(E\backslash a)\backslash a$  does not find any transitions to be removed anymore. As another example, consider  $a.(b.nil + b.nil)$  and  $a.b.nil + a.b.nil$ . It seems very reasonable that these two expressions could, or even should, be treated as equivalent. Syntactically speaking, this could be achieved by something like a distributivity law for prefixing (over choice). However, one needs to be careful, because we *do not* want a general left-distributivity law to hold in the process algebra context. The argument is supplied by the following two expressions:

$$a.(b.nil + c.nil) \quad \text{and} \quad a.b.nil + a.c.nil.$$

There is a context in which they behave differently, viz.  $((\bullet \hat{a}.\hat{c}.nil)\backslash a)\backslash b\backslash c$ . Put first  $a.(b.nil + c.nil)$  in place of the bullet, and observe that there is no deadlock: two  $\tau$ ’s can be executed in succession, and the execution terminates normally.

Then put  $a.b.\text{nil} + a.c.\text{nil}$  in place of the bullet and observe that now there *can* be a deadlock. It arises when the  $\hat{a}$  is synchronised with the  $a$  in front of the  $b$ , rather than with the  $a$  in front of the  $c$ , and executed. If this is not seen immediately from the SOS rules, we would encourage the reader to draw the two nets and check that they differ in an additional  $\tau$ -labelled transition in the second net, which is the one creating the deadlock. We conclude that distributivity of prefix over choice (from the left) is not, in general, a good idea. Still, it seems that we *ought* to have some means of equating expressions, because there are just too many of them around: It is easy to find more examples that one might intuitively call equivalent, e.g.,  $a.\text{nil}$  and  $a.\text{nil} + a.\text{nil}$ , or even (we are talking ‘interleaving’)  $a.\text{nil} \parallel b.\text{nil}$  and  $a.b.\text{nil} + b.a.\text{nil}$ .

The discussion suggests that a *behavioural*, rather than a structural, or a syntactic, equivalence relation is sought for. One of the fundamental contributions of early CCS [24, 27] is the development of a very important notion of this kind, called *bisimulation equivalence*, or just *bisimulation*. The basic idea is to relate sets of states of the two processes that are subject to comparison, in such a way that from two related states, if one of them can make a move, the other one can make a similar move such that the new states are again related. By a ‘state’ of a process, we mean any process that can be reached after a number of moves (including the initial process which can be reached after zero moves). The set of all reachable states can be viewed as the nodes of a directed graph, whose arrows correspond to the moves from one state to another. This edge-labelled graph is usually called the *transition system* (of the process). (The terminology is due to [21].) If an initial state is present (as is usually the case), we may use a special symbol for it – below, we use a circle which is not entirely black. These notions are actually similar to the *reachability graphs*, which comprises the set of reachable markings of a Petri net in graphical form.



**Fig. 7.** Transition systems for  $a.(b.\text{nil} + c.\text{nil})$  and  $a.b.\text{nil} + a.c.\text{nil}$ .

Figure 7 explains why no bisimulation can be found between  $a.(b.\text{nil} + c.\text{nil})$  and  $a.b.\text{nil} + a.c.\text{nil}$ : On the right-hand side, neither state  $s'_{21}$  nor state  $s'_{22}$  allow *both*  $b$  and  $c$  actions to occur, and thus neither of them corresponds to state  $s_2$ . Intuitively, the moment of choice between  $b$  and  $c$  is pre-poned in the process  $a.(b.\text{nil}) + a.(c.\text{nil})$ , choosing here between the first or the second  $a$  already determines the choice between  $b$  and  $c$ . Note that, by contrast, a bisimulation *can* be found between the two expressions  $a.((b.\text{nil}) + (b.\text{nil}))$  and  $a.(b.\text{nil}) + a.(b.\text{nil})$ . Their two transition systems are exactly the same as the ones

shown in Figure 7, except that the  $c$  is on both sides replaced by a  $b$ . In that case,  $\{(s_1, s'_1), (s_2, s'_{21}), (s_2, s'_{22}), (s_3, s'_3), (s_4, s'_4)\}$  is a bisimulation.

Because bisimulation is defined on transition systems, rather than on expressions or nets, it can be defined in a similar way for any formal model from which a transition system can be derived. A transition system over a given set of symbols  $\Sigma$  is a pair  $(S, \rightarrow)$  such that  $S$  is some set of states (usually with a designated initial state) and  $\rightarrow$  is a set of triples  $(s, x, \tilde{s})$ , such that  $s$  and  $\tilde{s}$  are states and  $x$  is a symbol from  $\Sigma$ . A triple  $(s, x, \tilde{s})$  indicates that a move from state  $s$  to state  $\tilde{s}$  (not necessarily different from  $s$ ) is possible by executing symbol  $x$ . One can also write  $s \xrightarrow{x} \tilde{s}$  instead of  $(s, x, \tilde{s}) \in \rightarrow$ . For example, the left-hand side of Figure 7 shows a transition system over  $\Sigma = \{a, b, c\}$  with four states  $S = \{s_1, s_2, s_3, s_4\}$  and three triples:  $(s_1, a, s_2)$ ,  $(s_2, b, s_3)$  and  $(s_2, c, s_4)$ . Note that a transition-labelled Petri net gives rise to two transition systems: (i) its reachability graph where the set of symbols is the set of transitions, and (ii) the graph obtained from (i) by replacing each transition with its label (the set of symbols is now the set of transition labels).

Let us have a look at the formal definition of bisimulation: Suppose  $(S_1, \rightarrow_1)$  and  $(S_2, \rightarrow_2)$  are two transition systems with initial states  $s_{01}$  and  $s_{02}$ , respectively. Then a relation  $\beta \subseteq S_1 \times S_2$  is called a bisimulation *if and only if* the initial states are related by  $\beta$  (i.e.,  $(s_{01}, s_{02}) \in \beta$ ), and whenever two states are related, then the moves in  $(S_1, \rightarrow_1)$  can be mirrored by moves in  $(S_2, \rightarrow_2)$  and vice versa, i.e., for each pair  $(s_1, s_2) \in \beta$  we have:

- If  $s_1 \xrightarrow{z} s'_1$ , then there is  $s'_2 \in S_2$  such that  $s_2 \xrightarrow{z} s'_2$  and  $(s'_1, s'_2) \in \beta$ .
- If  $s_2 \xrightarrow{z} s'_2$ , then there is  $s'_1 \in S_1$  such that  $s_1 \xrightarrow{z} s'_1$  and  $(s'_1, s'_2) \in \beta$ .

We have been careful to give the  $\Sigma$  in the definition of a transition system a different name than  $A$ , even though one usually takes  $\Sigma = A$  or  $\Sigma = A \cup \{\tau\}$ , in the case of basic CCS. The resulting notion of bisimulation is called *strong bisimulation*. Often, one wants to ‘neglect  $\tau$ -moves’, whenever possible. For instance, the processes  $a.\tau.\text{nil}$  and  $a.\text{nil}$  are not strongly bisimilar, because the  $\tau$  is treated like a normal action. It is possible (and often desirable) to consider a weaker notion of bisimulation that ignores internal  $\tau$ -moves whenever this not lead to a loss of vital semantical information. For instance, it would be too hasty to equate  $a.\text{nil} + b.\text{nil}$  and  $a.\text{nil} + \tau.b.\text{nil}$ , because executing the  $\tau$  implies a ‘silent’ but firm commitment to executing  $b$  (rather than  $a$ ) as the next action. A version of bisimulation called *weak bisimulation* [25] arises out of the previous definition thus:  $\beta$  is called a weak bisimulation between the two transition systems *if and only if* the initial states are related by  $\beta$ , and whenever two states are related, then the moves in  $(S_1, \rightarrow_1)$  can be mirrored by (possibly empty) *sequences* in  $(S_2, \rightarrow_2)$  and vice versa, i.e., for each pair  $(s_1, s_2) \in \beta$  we have:

- If  $s_1 \xrightarrow{z} s'_1$ , then there is  $s'_2 \in S_2$  such that  $s_2 \xrightarrow{w} s'_2$  and  $(s'_1, s'_2) \in \beta$ , where:  $w$  is a sequence of  $\tau$ ’s if  $z = \tau$ , and  $z$  surrounded by sequences of  $\tau$ ’s if  $z \neq \tau$ ; and  $s_2 \xrightarrow{w} s'_2$  means that  $s'_2$  is reached from  $s_2$  by successive executions of the actions in  $w$ .

- If  $s_2 \xrightarrow{z} s'_2$ , then there is  $s'_1 \in S_1$  such that  $s_1 \xrightarrow{w} s'_1$  and  $(s'_1, s'_2) \in \beta$ , where  $w$  is similar as above.

Let us write  $E \cong E'$  and  $E \cong_w E'$  if  $E$  and  $E'$  are *strongly bisimilar* and, respectively, *weakly bisimilar*, i.e., if there exists a strong (weak, respectively) bisimulation between their transition systems. It is fairly clear that these relations are indeed equivalence relations, i.e., they are reflexive, symmetric, and associative. Obviously, a strong bisimulation is also a weak bisimulation, so that  $E \cong E'$  implies  $E \cong_w E'$  (i.e.,  $\cong \subseteq \cong_w$ ). For instance,  $a.b.\text{nil}$  and  $a.\tau.b.\text{nil}$  are weakly bisimilar (when checking this be aware that the empty sequence is also a sequence), since the two intermediate states in the latter can be related to the single intermediate state of the former, but  $a.\text{nil} + b.\text{nil}$  and  $a.\text{nil} + \tau.b.\text{nil}$  are not.

Strong and weak bisimilarity of process expressions are actually quite strong equivalence notions, short of actual syntactic equality of process terms. There are also much weaker notions of equality between processes. A particularly important one is *trace equality*, defined as follows:  $E$  and  $E'$  are trace-equivalent if the set of all sequences of possible moves of  $E$  and  $E'$  (i.e., their interleavings) coincide. It is easy to prove that whenever  $E$  and  $E'$  are (strongly or weakly) bisimilar, then  $E$  and  $E'$  are also trace equivalent. However,  $a.(b.\text{nil} + c.\text{nil})$  and  $a.b.\text{nil} + a.c.\text{nil}$  are trace equivalent without being bisimilar. Between bisimulation and trace equality, there is a spectrum of interesting equivalence notions, as described in [14]. Although bisimulation is a strong equivalence notion, there are still stronger ones. A next to useless one, already mentioned, is syntactic equality. Another one is *transition system isomorphism*, defined as follows:  $E$  and  $E'$  are *ts-isomorphic* if there exists an isomorphism between their transition systems. For instance,  $E + F$  and  $F + E$  are ts-isomorphic, but  $a.a.\text{nil}$  and  $a.\text{nil} \mid a.\text{nil}$  are not (but they are bisimilar). There also are desirable identifications that require only a slight strengthening of the notion of bisimulation. For instance, one might wish to distinguish  $(b.a.\text{nil}) \backslash a$  and  $b.\text{nil}$ , because (as discussed earlier on) the latter, but not the former, terminates properly; and yet, they are (strongly and weakly) bisimilar.

Now that we have defined strong and weak bisimilarity and sketched their characteristics, some real *calculations* can be started, in particular, we may now set out to *prove* a host of equivalences. For example, the following algebraic properties of the choice operator become actually provable:

$$\begin{array}{lll}
 E + \text{nil} & \cong E & (\text{nil is neutral w.r.t. choice, cf. the next line}) \\
 E + F & \cong F + E & (\text{choice is symmetric}) \\
 (E + F) + G & \cong E + (F + G) & (\text{choice is associative}),
 \end{array}$$

for all  $E$ ,  $F$ , and  $G$ . Similarly, the composition operator  $\mid$  is symmetric, associative, and has  $\text{nil}$  as a neutral element, with respect to strong bisimulation. The associativity of  $\mid$  requires a trifle more thought than the other properties. It holds because of the careful design of basic CCS. In many practical cases, a composition of two processes is immediately followed by restriction, such as in  $(E \mid F) \backslash a$ . Suppose that we had defined  $\mid$  in such a way that it *included* restric-

tion. Then we would have lost associativity. To see this, compare the following two expressions:

$$(((a.\text{nil} \mid a.b.\text{nil}) \backslash a) \mid \hat{a}.\text{nil}) \backslash a \quad \text{and} \quad (a.\text{nil} \mid ((a.b.\text{nil} \mid \hat{a}.\text{nil}) \backslash a)) \backslash a .$$

The expression on the left-hand side is bisimilar to  $\text{nil}$ , while the expression on the right-hand side can make a  $\tau$ -move followed by a  $b$ -move, and therefore, associativity (of the hypothetical composition operator) does not hold. The reason why such an example cannot occur for the basic CCS composition operator, is that all conjugate pairs are still around after synchronisation, unless they are *explicitly* restricted away by a *different* operator.

Restriction, too, has several useful algebraic properties, such as:

$$\begin{aligned} (E \backslash a) \backslash b &\cong (E \backslash b) \backslash a && \text{(restriction is symmetric)} \\ (E + F) \backslash a &\cong (E \backslash a) + (F \backslash a) && \text{(restriction distributive over choice)} \end{aligned}$$

and more (for instance, the reader might wish to find out under which condition(s) restriction distributes over composition).

We also have some *congruence* properties, that is, we may prove that  $\cong$  is preserved under some operations. Just as one particular example, the following holds true:

$$E \cong E' \implies E \mid F \cong E' \mid F ,$$

for all processes  $E$ ,  $E'$  and  $F$ . There are many more algebraic properties of this kind, but this is not the right place to list them all – well, if ‘listing them all’ is at all possible. Suppose that we had a finite set of such properties, such that for *any* two expressions  $E$  and  $E'$ , if  $E$  and  $E'$  are bisimilar, then this can be proved using only those properties. If such a set could be found, then we would call bisimilarity *finitely axiomatisable*. However, it has been shown that bisimilarity is *not* finitely axiomatisable for process algebras including basic CCS [30]. Thus, the list of ‘interesting algebraic properties’ analogous to the above ones, has to be virtually endless. Let us just list two of them. The first one gives a general property for weak bisimulation:

$$a.(E + \tau.F) + a.F \cong_w a.(E + \tau.F) ,$$

which is taken from [25]. It is a good exercise to prove this for simple  $E$  and  $F$ , for instance  $b.\text{nil}$  and  $c.\text{nil}$ , and also to check why  $\cong_w$  cannot be replaced here by the stronger  $\cong$ . Another property is the *expansion property*. It allows composite expressions to be ‘developed’ into large sums (comprising the possible ‘first actions’ that can be executed in the composite expression followed by smaller composite expressions). We give here only an application of this property:

$$\begin{aligned} a.\text{nil} \mid \hat{a}.\text{nil} &\cong a.(\text{nil} \mid \hat{a}.\text{nil}) + \hat{a}.(a.\text{nil} \mid \text{nil}) + \tau.(\text{nil} \mid \text{nil}) \\ &\cong a.\hat{a}.\text{nil} + \hat{a}.a.\text{nil} + \tau.(\text{nil} \mid \text{nil}) \\ &\cong a.\hat{a}.\text{nil} + \hat{a}.a.\text{nil} + \tau.\text{nil} . \end{aligned}$$

In this chain of bisimilarities, the first two come from the expansion property, and the last one from  $\text{nil} \mid \text{nil} \cong \text{nil}$ . We have managed to reduce an expression with  $\mid$  operator into one without. The expansion property can *always* be used in this way, showing that (at least with respect to bisimulation as defined above) the  $\mid$  operator is redundant for recursion-free expressions<sup>1</sup>. This is strongly related to the fact that, so far, we *only* consider interleaving semantics. In fact, the validity of the expansion property can be viewed as a direct *formalisation of interleaving* (concurrency can be replaced by nondeterministic interleaving).

We have called the above equalities ‘properties’ rather than ‘laws’, because they are consequences of the definitions, rather than being postulated. It is possible to turn the approach upside down, as exemplified by [3]. Let us start with just the set  $A$  of actions (and possibly  $\tau$ ) and simply *postulate* a set of reasonable algebraic equations for a set of operators that are interesting in that they make sense for a process algebra, such as ‘+’ for choice (as before) and ‘;’ for sequential composition. We quote the first five postulates (also called *axioms*) from [3]:

$$\begin{array}{lll} p + q = q + p & (p + q) + r = p + (q + r) & p + p = p \\ (p + q); r = p; r + q; r & (p; q); r = p; (q; r) & . \end{array}$$

The axioms in the first row are already known from basic CCS’s choice operator, except that here,  $p$ ,  $q$  and  $r$  range over an unknown ‘set of processes’, and the equality is not a concrete one, but a postulated one (which may, or may not<sup>2</sup>, later be substituted by a concrete one). The next one is the ‘wrong’ distributivity, or rather the ‘right’ one, both by nature and since it can safely be assumed to hold (as opposed to left distributivity, which was rejected earlier). Note that it cannot even be formulated in basic CCS, since a compound process such as  $p + q$  cannot be placed in front of sequential composition there. For the same reason, associativity of sequential composition (the last one in the above list of axioms) cannot be formulated in basic CCS. Thus, it is already clear that with the above axioms, one aims at a process algebra that is somewhat different from basic CCS. Eventually, through the development and after adding more and more axioms, processes satisfying these axioms can be constructed via terms and graphs as transition systems, i.e.: transition systems are a *model* of those equations, after the appropriate operations have been defined on them and bisimilarity is taken as equality.

What is the advantage of postulating axioms and looking for models that satisfy them, over defining a particular model such as basic CCS and proving properties about it? Well, sometimes one wishes to be somewhat at liberty with the adopted axioms and, in particular, wants to investigate their interplay. Searching for interesting models and proving that they satisfy the axioms could,

<sup>1</sup> However, the practical use of this insight is very limited indeed; for example, [3] gives the expansion of  $(a.a.a.\text{nil} \mid b.b.b.\text{nil}) \mid c.c.c.\text{nil}$ . It takes two full pages in small print!

<sup>2</sup> For instance, we might be as careless as to postulate *contradictory* laws.

however, be a none too trivial matter. There are also other angles by which process algebras can be viewed and classified. For instance, one might be interested in categorising them in terms of different SOS semantics formats [1]. Another approach is *parametric*, in the sense that one considers a whole class of concrete process algebras and proves properties that hold for each one. We will turn to one of these approaches in the next section.

## 5 B.TOYA: A Basic Toy Algebra

We are now prepared to consider process algebras with operations that slightly deviate from, or generalise, those of basic CCS. We'll discuss another concrete algebra and its semantics, which are designed specifically with their relationship to Petri nets in mind. Inevitably, through such a relationship, we need to consider Petri nets as *composable* entities, and the intended translation will be *compositional*, in the sense that the net of a composite process is the composition of the nets associated with the composed processes.

Let us start by decomposing the basic CCS composition operator. As described above, it consists of a first step (corresponding to forming the disjoint union of Petri nets) and a second step (calculating the synchronisations of whatever conjugate pairs can be found in the constituent processes, or respectively, their nets). Now suppose that we separated disjoint union from synchronisation. In CCS, composition forces synchronisation for *all* pairs  $a, \hat{a}, b, \hat{b}$ , etc., in a single step. By analogy to the unary restriction operator, let us contemplate a *unary* operator (parametrised by  $a$  and written in postfix style) that effects synchronisation just on pairs  $a, \hat{a}$ . Let us denote that operator  $\text{sy } a$ , and let us use  $\text{rs } a$ , instead of the previous  $\backslash a$ , to denote restriction. Suppose that we can prove the following properties for the  $\text{sy}$  operator:

$$\begin{aligned} E \text{ sy } a &\cong E \text{ sy } \hat{a} && \text{(insensitivity to conjugation)} \\ (E \text{ sy } a) \text{ sy } a &\cong E \text{ sy } a && \text{(idempotence)} \\ (E \text{ sy } a) \text{ sy } b &\cong (E \text{ sy } b) \text{ sy } a && \text{(symmetry)}, \end{aligned}$$

with some (strong) equivalence  $\cong$ . This would, in fact, make synchronisation algebraically quite similar to restriction (though almost 'opposite' in meaning), since restriction satisfies the same properties as well. The three properties allow us, by the way, to extend both operators uniquely to finite sets  $B$  of action names,  $\text{sy } B$  and  $\text{rs } B$ , and even to  $B = A$ , i.e., the entire set of action names. Using  $\text{sy } A$  then comes quite close to the original CCS intention behind the (second step in the) composition operator.

The point of the last paragraph was not to advertise yet another synchronisation operator, but to argue that the set of (elementary) manipulations in basic CCS (and, as it turns out, in similar concrete process algebras as well) can be divided – in terms of the nets associated with expressions – into two classes:

- Those manipulating *places*. Examples are choice, prefixing (or, more generally, sequential composition), and disjoint parallel composition (i.e., the first step in CCS's  $\mid$  operator).



- Those manipulating *transitions*. Examples are restriction and synchronisation (i.e., the second step in CCS's  $\mathbf{I}$  operator).

Let us first have a look at the latter class. Clearly, transitions will be manipulated depending on their labels. In case of restriction  $\mathbf{rs} \ a$ , for example, all transitions labelled  $a$  or  $\hat{a}$  are simply removed. In case of synchronisation  $\mathbf{sy} \ a$ , all transitions are left as they are, but some new ones are added, which can be viewed as the sums of two old ones (one labelled by  $a$ , the other by  $\hat{a}$ ).

The above suggests to search for a uniform and general way of describing these transition-(label-)based operations all at once. In [6], *general relabellings* were introduced for this purpose. The idea is that a set of transitions (by virtue of their labels) can be specified to give rise to a new transition, which is the net-theoretic sum of the old ones. Because different transitions may have the same label, we need to consider *multisets of labels* as the arguments of general relabelling. Therefore, a general relabelling  $\rho$  is defined as a relation  $\rho \subseteq \mathbf{mult}(\Sigma) \times \Sigma$ , where in case of basic CCS,  $\Sigma$  could be  $A \cup \{\tau\}$ , and  $\mathbf{mult}(\Sigma)$  denotes the set of multisets whose elements are in  $\Sigma$ . As an example, if we wished to describe restriction  $\mathbf{rs} \ a$  to have the same effect as restriction  $\backslash a$  in basic CCS, we could consider:

$$\rho_{\mathbf{rs} \ a} = \{(\{z\}, z) \mid z \in A \setminus \{a, \hat{a}\}\}.$$

That is, all transitions with labels not in  $\{a, \hat{a}\}$  are left intact, while those with label  $a$  or  $\hat{a}$  have no entry in  $\rho_{\mathbf{rs} \ a}$  and are therefore omitted. As another example, synchronisation  $\mathbf{sy} \ a$  can be described by:

$$\rho_{\mathbf{sy} \ a} = \{(\{z\}, z) \mid z \in A \cup \{\tau\}\} \cup \{(\{a, \hat{a}\}, \tau)\}.$$

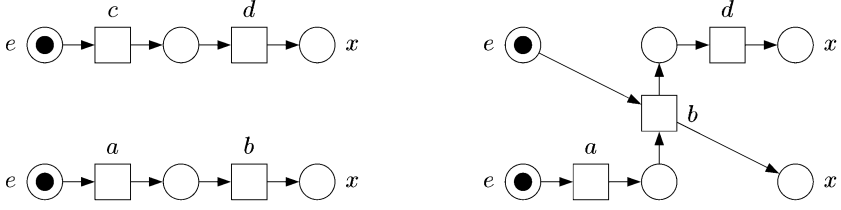
That is, *all* transitions are left intact, but some new  $\tau$ -labelled ones are added, as they arise from pairs of transitions, of which one is labelled by  $a$  and the other by  $\hat{a}$ .

For the sake of using the concept of general relabelling in a slightly non-standard way, let us design a ‘toy’ process algebra with a unary synchronisation operator that is parametrised by three action names,  $y, y', z \in \Sigma$ , and that takes any pair  $y, y'$ , creates a synchronised action  $z$  out of them (like previously the  $\tau$  out of  $a$  and  $\hat{a}$ ), and deletes all the  $y$ 's and  $y'$ 's afterwards. Denote this operator by  $\bullet[y, y' \rightarrow z]$ . Furthermore, in our toy algebra, we wish to have forward/backward symmetric choice, sequential, and parallel operators, the latter meaning disjoint parallelism. Basic TOYA is thus defined as follows (we assume  $y, y', z \in \Sigma$ ):

$$\mathbf{B.TOYA} : E ::= z \mid E[y, y' \rightarrow z] \mid E + E \mid E; E \mid E \parallel E,$$

where we have chosen the ‘ $\parallel$ ’ symbol for parallel composition, which is different from the composition in CCS, and the semicolon to indicate sequential composition, which is more general than prefixing in CCS.

By calling the operators forward/backward symmetric, we mean that their (intended) Petri nets should look symmetric, whether viewed from the entry



**Fig. 8.** Nets for  $((a;b) \parallel (c;d))$  and  $((a;b) \parallel (c;d))[b, c \rightarrow b]$ .

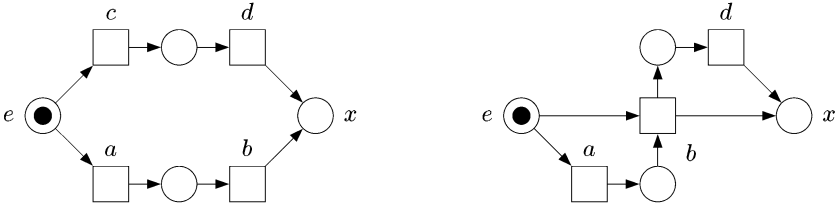
places or seen from the exit places (in Figure 2, only the nets shown on the left-hand side and on the right-hand side are forward/backward symmetric, the others are not). For instance, consider the expression:

$$((a;b) \parallel (c;d))[b, c \rightarrow b].$$

Its net *before* applying the  $\bullet[b, c \rightarrow b]$  operator is shown on the left-hand side of Figure 8, and the net *after* applying it is shown on the right-hand side of the figure. Consider, by contrast, the expression:

$$((a;b) + (c;d))[b, c \rightarrow b]$$

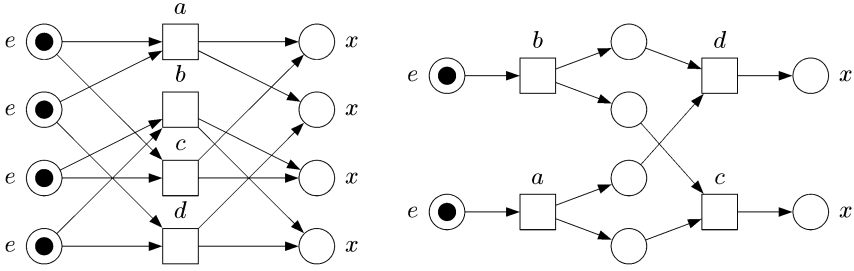
which is like the previous one, except that the  $\parallel$  has been replaced by a choice operator. Its net is shown on the right-hand side of Figure 9. Observe that it is intuitively correct that the  $b$  action in the middle can never occur, because the  $b$  in  $(a;b)$  requires  $a$  to occur first, while executing  $a$  disables  $c$ , which would be needed for the (resulting)  $b$  to occur.



**Fig. 9.** Nets for  $(a;b) + (c;d)$  and  $((a;b) + (c;d))[b, c \rightarrow b]$ .

Let us have a look at the net translation of B.TOYA expressions in which the parallel operator is nested, such as  $(a \parallel b) + (c \parallel d)$  and  $(a \parallel b); (c \parallel d)$ . In the former, as soon as  $a$  is chosen, neither  $c$  nor  $d$  can be chosen anymore. In the latter,  $c$  or  $d$  have to wait until both  $a$  and  $b$  have been completed. It is well-known how this can be achieved by means of Petri nets, and the two translations are shown in Figure 10. Both of these translations involve manipulations on the  $e$  and  $x$  places of the constituent nets, known as *place multiplication*. On the left-hand side of Figure 10, the  $e$  places of the first constituent,  $(a \parallel b)$ , are multiplied

with the  $e$  places of the second constituent,  $(c \parallel d)$ , and the same is done (for reasons of forward/backward symmetry) with their  $x$  places. On the right-hand side of Figure 10, the  $x$  places of the first constituent are multiplied with the  $e$  places of the second – as befits a sequential composition.



**Fig. 10.** Nets for  $(a \parallel b) + (c \parallel d)$  and  $(a \parallel b); (c \parallel d)$ .

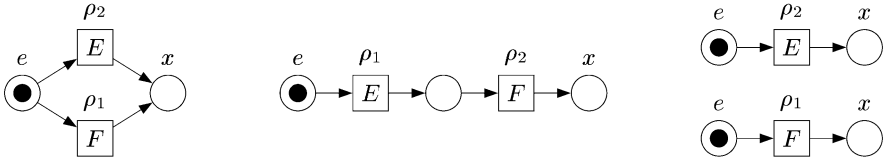
A systematic way of unifying place multiplication and its use in the Petri net semantics of process algebras is to use *transition refinement* [15]. In transition refinement, a transition in a Petri net is replaced by a whole (sub)net (i.e., a net which, after refinement, becomes a subnet). If the net by which transition  $t$  is replaced (called the *refining net*), carries  $e$  and  $x$  tags on some of its places, then one can combine the former with the input places of  $t$ , and the latter with the output places of  $t$ , pairwise, to create appropriate new places by place multiplication. If care is exercised, then properties of the resulting *refined* net can be inferred from properties of the refining net and the net in which  $t$  is embedded. In particular, the S-invariants of the refined net can be calculated from those of the original net and those of the refining net [6]<sup>3</sup>. Moreover, transition refinement is symmetric, i.e., refining the first transition  $t_1$  and then  $t_2$  is the same as refining the two transitions in the other order. Hence one may employ, without ambiguity, a *simultaneous refinement* of a set of transitions. And finally, for a vast class of cases, the safeness (1-boundedness) of the resulting net can be inferred from the safeness of the original net and the safeness of the refining nets.

Let us see how simultaneous transition refinement can help in unifying the treatment of choice and sequential composition, and also of disjoint concurrent composition. Figure 11 shows three nets which correspond to the operations just mentioned. Let us look first at the net for choice, on the left-hand side of Figure 11: Refining the transition labelled  $\rho_1$  by the net corresponding to  $(a \parallel b)$ , and simultaneously refining the transition labelled  $\rho_2$  by the net corresponding to  $(c \parallel d)$ , yields the net shown on the left-hand side of Figure 10. Let us then consider the net for sequence in the middle of Figure 11: Refining the transition labelled  $\rho_1$  by the net corresponding to  $(a \parallel b)$ , and simultaneously the transition

<sup>3</sup> This result is due to R. Devillers.

labelled  $\rho_2$  by the net corresponding to  $(c \parallel d)$ , yields the net shown on the right-hand side of Figure 10.

Thus, the two operations can quite similarly be described by simultaneous transition refinement. They are distinguished only by the original Petri net to which transition refinement is applied. Disjoint concurrent composition can also be described by simultaneous transition refinement, with the only distinction that the to-be-refined-net is the one shown on the right-hand side of Figure 11. Thus, these three nets can be viewed as binary *operator nets*: They each take two nets as arguments and create a new net out of them through simultaneous transition refinement.



**Fig. 11.** Operator nets for choice, sequence and parallel composition.

Actually, general relabellings can be integrated seamlessly into this idea of simultaneous transition refinement. For, suppose that every transition in an operator net is labelled by a general relabelling  $\rho$ . Then one can define simultaneous refinement in such a way that during its application, the respective relabellings are *also* carried out. In Figure 11, we have already provided every transition with a general relabelling  $\rho_i$  ( $i = 1, 2$ ), but since each of these operations does *not* create any new transitions, nor destroy old ones, every  $\rho_i$  should be defined as the ‘identity’,

$$\rho_i = \{(\{z\}, z) \mid z \in \Sigma\},$$

which leaves every transition intact, and unmodified. An operator net for the unary operation  $[y, y' \rightarrow z]$  has one transition and it carries, by contrast, the following general relabelling:

$$\rho_{[y, y' \rightarrow z]} = \{(\{v\}, v) \mid v \in \Sigma \setminus \{y, y'\}\} \cup \{(\{y, y'\}, z)\},$$

as shown on the left-hand side Figure 12. Finally, we need a standard Petri net for each of the constants of the algebra, i.e., for an action  $z \in \Sigma$ . It is shown on the right-hand side of Figure 12.



**Fig. 12.** Operator net for  $E[y, y' \rightarrow z]$ , and Petri net for constant  $z$ .

As an example, we consider the systematic derivation of the net corresponding to the expression  $((a; b) \parallel (c; d))[b, c \rightarrow b]$ , cf. Figure 8. To get this net, we will proceed uniformly by structural induction: first, we create four nets for  $a$ ,  $b$ ,  $c$ ,  $d$  (using four times the right-hand side of Figure 12). Then, we use simultaneous refinement twice on the operator net for sequential composition, to get nets for  $(a; b)$  and for  $(c; d)$ , respectively. We use these two to simultaneously refine the operator net for parallel composition and derive a net for  $((a; b) \parallel (c; d))$ . Finally, we refine the net shown on the left-hand side of Figure 12 by the latter (which involves an application of  $\rho_{[y, y' \rightarrow z]}$ ), to get the end result shown on the right-hand side Figure 8.

This approach can be called parametric, because we could, if we so wished, consider many other operators of the algebra, not just the ones shown in Figures 11 and 12. However, it is surely not a very good idea to use just *any* Petri net in the role of an operator net. In particular, we wish to still be able to define decent SOS rules for the process algebra(s) so obtained.

## 6 SOS Rules for B.TOYA

The standard SOS approach does not match exactly the idea of marking changes in a Petri net, because a net's structure is *not* changed by transition occurrences. However, it is possible to modify SOS in such a way that it does not affect the underlying structure of a process algebraic expression. To this end, we may introduce markings into process algebraic expressions. A (perhaps deceptively) simple (but, as we will see, sufficient) way of doing this is by overbarring and/or underbarring subexpressions. For example, we may write  $\overline{a}$  to describe the fact that a token is present before  $a$ , using which,  $a$  may be executed. Similarly,  $\underline{a}$  can be used to describe the fact that a token is present after  $a$  and it *may* have arrived there by executing  $a$  (although it might have arrived there in some other way).

Let us look at all possible over- and underbarrings of all subexpressions of a simple expression such as  $a; b$ , i.e.:  $\overline{a}; \overline{b}$ ,  $\overline{a}; b$ ,  $\underline{a}; b$ ,  $a; \overline{b}$ ,  $a; \underline{b}$  and  $\underline{a}; b$  (there are no other syntactic possibilities). But these are more than enough: The first describes the initial marking of the entire expression  $a; b$ . The second describes the initial marking of its first part,  $a$ . The third describes the intermediate marking reached after executing  $a$ . The fourth describes the initial marking of its second part,  $b$ . The fifth and sixth describe the marking reached after executing  $b$ , and the marking reached after executing the entire expression, respectively. These expressions (that arise from B.TOYA expressions by over- or underbarring subexpressions) will be called *dynamic expressions* (as opposed to non-barred ones, which will be called *static*).

Clearly, due to the nature of sequential composition, some of the above should be identified, because they correspond to the same marking in the associated net:  $\overline{a}; \overline{b} \equiv \overline{a}; b$ ,  $\underline{a}; b \equiv a; \overline{b}$  and  $a; \underline{b} \equiv \underline{a}; b$ . We use here the symbol  $\equiv$  to mean (loosely speaking): 'giving rise, in the intended semantics, to the same marked Petri net'. Let us have a closer look at  $\underline{a}; b \equiv \overline{b}; a$ : Both dynamic

expressions correspond to the intermediate marking reached after executing the transition labelled  $\rho_1$  (which is the one corresponding to  $a$  in the expression  $a;b$ ) in the operator net for sequential composition, see Figure 11 (middle). In general, simply by looking at the operator net (and not worrying about how arbitrarily complex a concrete sequentially composed expression may become after refinement), we can find all the identifications to be made by  $\equiv$ . Before we give the list of identifications for B.TOYA, we extend the syntax to include dynamic expressions. We denote them by  $G$  and  $H$ , rather than  $D$ ,  $E$  or  $F$ , which continue to stand for static expressions:

$$\text{DB.TOYA : } G ::= \overline{E} \mid \underline{E} \mid G[y, y' \rightarrow z] \mid G + E \mid E + G \mid G; E \mid E; G \mid G \parallel H.$$

Note how the fact that choice is distinguished from concurrent composition is reflected in this syntax: in a dynamic expression, only one of the two sides of a choice may be dynamic, while both sides of a concurrent composition are dynamic. Here are the general identifications of DB.TOYA expressions, as derived from the corresponding operator nets:

- (1) (a)  $\overline{E[y, y' \rightarrow z]} \equiv \overline{E}[y, y' \rightarrow z]$  , (b)  $\underline{E[y, y' \rightarrow z]} \equiv \underline{E[y, y' \rightarrow z]}$
- (2) (a)  $\overline{E + F} \equiv \overline{E} + F \equiv E + \overline{F}$  , (b)  $\underline{E + F} \equiv E + \underline{F} \equiv \underline{E + F}$
- (3) (a)  $\overline{E; F} \equiv \overline{E}; F$  , (b)  $\underline{E; F} \equiv E; \underline{F}$  , (c)  $E; \underline{F} \equiv \underline{E; F}$
- (4) (a)  $\overline{E \parallel F} \equiv \overline{E} \parallel \overline{F}$  , (b)  $\underline{E \parallel F} \equiv \underline{E} \parallel \underline{F}$  .

Line (3) is the general version of what has already been explained on an example. Line (2a) states that the initial marking of a choice initialises both of its constituents (but only one of them at the same time). Line (4a) states, by contrast, that the initial marking of a concurrent composition initialises both of its constituents at the same time. The other identifications have similar justifications.

We need a general context rule: suppose that  $\mathcal{C}(\bullet)$  is some valid dynamic context, i.e., that in place of the  $\bullet$  some dynamic expression  $G$  could be written, such that the resulting string  $\mathcal{C}(G)$  is again a valid dynamic expression. Then,

$$(\text{CR}) \quad \text{If } G \equiv H \text{ , then } \mathcal{C}(G) \equiv \mathcal{C}(H) \text{ .}$$

To mimic the marking changes in the (intended) Petri net semantics, we need an equivalent of the Petri net transition rule in terms of dynamic expressions. Let us generalise this by defining *concurrent occurrences*, i.e., *steps*, of actions. Thus, for instance, in Figure 10, on the left-hand side, a step  $\{a, b\}$  can occur, since  $a$  and  $b$  are concurrently enabled, and, for similar reasons, a step  $\{c, d\}$ , but also a step  $\{a\}$  or a step  $\{b\}$  (or also the empty step, making no marking change), but *not*, of course, a step  $\{a, c\}$ . The rules are provided by the next list:

$$\begin{array}{ll}
(\alpha) \quad \overline{a} \xrightarrow{\{a\}} \underline{a} & (A) \quad \frac{G \xrightarrow{\gamma+k \cdot \{y, y'\}} H, k \in \mathbb{N}}{G[y, y' \rightarrow z] \xrightarrow{\gamma+k \cdot \{z\}} H[y, y' \rightarrow z]} \\
(B) (1) \quad \frac{G \xrightarrow{\gamma} H}{G + E \xrightarrow{\gamma} H + E} & (2) \quad \frac{G \xrightarrow{\gamma} H}{E + G \xrightarrow{\gamma} E + H} \\
(C) (1) \quad \frac{G \xrightarrow{\gamma} H}{G; E \xrightarrow{\gamma} H; E} & (2) \quad \frac{G \xrightarrow{\gamma} H}{E; G \xrightarrow{\gamma} E; H} \\
(D) \quad \frac{G \xrightarrow{\gamma} G', H \xrightarrow{\delta} H'}{G \parallel H \xrightarrow{\gamma+\delta} G' \parallel H'}, &
\end{array}$$

where we write  $\xrightarrow{\gamma}$  for a step  $\gamma \in \text{mult}(\Sigma)$ , and  $\gamma$  and  $\delta$  are, in general, multisets of actions<sup>4</sup>. Also,  $k \cdot \{y, y'\}$  denotes the multiset where  $y$  and  $y'$  have multiplicity  $k$  (and all other actions multiplicity 0). Note that  $(\alpha)$  is the only rule which allows something (namely,  $a$ ) to actually ‘happen’. All other rules are only ‘context rules’, in the sense that they allow to infer some global occurrence from something that happens locally. This is reinforced by the observation that, in fact, the rules for choice and for sequential composition, (B) and (C), are *exactly* the same, differing only in the operator used. In fact, sequence and choice are distinguished *only* by their different  $\equiv$  relations in lines (2) and (3). Concurrent composition and synchronisation  $[y, y' \rightarrow z]$  again have the same context rule, and are distinguished only by their different shape in terms of the syntax of DB.TOYA.

The rule for choice may be contrasted with the corresponding rule for basic CCS, given in section 2. The main difference is (and that is, of course, what was desired) that the underlying structure of a dynamic expression (that is, the structure obtained by removing all over- and underbars) does not change during the execution of a choice.

Let us see how one would derive the step sequence  $\xrightarrow{\{a\}\{b\}}$ , which is (or should be, by inspecting the intended Petri net semantics shown on the right-hand side Figure 8) derivable as a valid execution from the initial state of the expression:

$$((a; b) \parallel (c; d))[b, c \rightarrow b].$$

In the derivation that follows, we list on the right-hand side the rules used in each derivation step.

$$\begin{aligned}
\overline{((a; b) \parallel (c; d))[b, c \rightarrow b]} &\equiv \overline{((a; b) \parallel (c; d))}[b, c \rightarrow b] \quad (1a) \\
&\equiv \overline{((a; b) \parallel (c; d))}[b, c \rightarrow b] \quad (4a, \text{CR}) \\
&\equiv ((\overline{a}; b) \parallel (\overline{c}; d))[b, c \rightarrow b] \quad (2\text{-}3a, \text{CR})
\end{aligned}$$

<sup>4</sup> They have to be multisets, rather than a sets, since two transitions with the same label could occur concurrently.

$$\begin{aligned}
 & \xrightarrow{\{a\}} ((\underline{a}; b) \parallel (\bar{c}; d))[b, c \rightarrow b] \quad ( \alpha, C1, D, A ) \\
 & \equiv ((a; \bar{b}) \parallel (\bar{c}; d))[b, c \rightarrow b] \quad ( 3b, CR ) \\
 & \xrightarrow{\{b\}} ((a; \underline{b}) \parallel (\underline{c}; d))[b, c \rightarrow b] \quad ( 2 \cdot \alpha, C2, C1, D, A ) \\
 & \dots
 \end{aligned}$$

In the line with  $\xrightarrow{\{a\}}$ , rule (D) has been applied with  $\gamma = \{a\}$  and  $\delta = \emptyset$ , and rule (A) with  $\gamma = \{a\}$  and  $k = 0$ . In the line with  $\xrightarrow{\{b\}}$ , rule (D) has been applied with  $\gamma = \{b\}$  and  $\delta = \{b\}$ , and rule (A) with  $\gamma = \emptyset$  and  $k = 1$ .

For historic (and also self-explanatory) reasons, semantics such as these SOS rules are called *operational*, and semantics such as given by the Petri net translation (through operator nets and simultaneous transition refinement, cf. section 5) are called *denotational*, and it is desirable to prove an equivalence between them (see, e.g., [26]). In the case of B.TOYA, we have a strong equivalence: *consistency*, i.e., every step sequence derivable by the SOS rules can also be derived in the net by means of the transition rule, and *completeness*, i.e., for every step sequence derivable in the net, there is *some* possibility of deriving it from the SOS rules. In proving these equivalence properties, it is essential that all operator nets satisfy certain appealing and desirable properties (for instance, 1-safeness, but also, a special property called *factorisability* [6]), and it is also essential that the relationship between the process algebra on the one hand, and the Petri nets on the other hand, is tight and one-to-one. In particular, it may be dangerous to use intermediate  $\tau$  transitions too liberally. For instance, the translation of choice shown on the right-hand side of Figure 2 has to be rejected.

## 7 Recursion

From the theory of computability, it is known that Turing machines can be simulated by 2-stack pushdown automata. Using basic CCS with recursion, an arbitrary number of stacks can be modelled. Thus – and this is how an argument in [25] goes – basic CCS with recursion is Turing-powerful. Therefore, it is impossible to find finite 1-safe (basic, i.e., place/transition) Petri nets corresponding to basic CCS expressions in general, since such nets have the computing power of finite automata. Even when the restriction to 1-safeness is lifted, we will not be able to find a translation to finite nets, because finite, unbounded place/transition nets are still not Turing-powerful, even though their computing power extends beyond that of finite automata [29]. If a finite Petri net translation is desired, one needs to use high-level Petri nets. A translation of basic CCS and TCSP into finite high-level nets has been described in [32].

In this section, we will show what may happen if the *finiteness* restriction, rather than the 1-boundedness restriction, is lifted, i.e., we will search for a translation of basic CCS with recursion into (possibly infinite) nets. And we will strive to keep the other restriction, 1-safeness, intact. Actually, rather than basic CCS, we will consider B.TOYA, augmented by a facility to express recursion:



$$\text{R.TOYA} : E ::= a \mid E[y, y' \rightarrow z] \mid E + F \mid E; F \mid E \parallel F \mid X ,$$

where every variable  $X$  is, by definition (and as in basic CCS), associated with a defining equation  $X \stackrel{\text{df}}{=} E$ . In such a defining equation,  $E$  is called the *body*. Since basic CCS can essentially be expressed in R.TOYA, save for the differences in synchronisation and parallel composition discussed above, the explanation of R.TOYA below could (albeit with some effort) be carried over to basic CCS. With the  $X \stackrel{\text{df}}{=} E$  device, we can write mutually recursive systems of equations, such as  $X_1 \stackrel{\text{df}}{=} a; X_2$  &  $X_2 \stackrel{\text{df}}{=} X_2 \parallel X_1$ . However, there are standard (well, almost standard) ways of reducing the number of equations in such systems, and we will concentrate, in the following, on just a single variable,  $X$ , and its unique defining equation,  $X \stackrel{\text{df}}{=} E$ .

For instance, let us consider the equation  $X \stackrel{\text{df}}{=} a; X$ , with body  $a; X$ . Clearly, we should be able to derive sequences of  $a$ -moves of arbitrary length from the initial state of  $X$ , but how can this be done formally? Well, let us start as follows:

$$\overline{X} \xrightarrow{\emptyset} \overline{a; X} \equiv \overline{a}; X \xrightarrow{a} \underline{a}; X \equiv a; \overline{X}.$$

In this derivation, all steps except the first one are applications of ideas already mentioned. The second and the last one, for instance, are applications of rules (3a) and (3b), which belong to the operator net for sequential composition. The first step,  $\overline{X} \xrightarrow{\emptyset} \overline{a; X}$ , is an application of the *recursion principle*, which states the following:

In any dynamic context, a recursion variable  $X$  may be (syntactically) rewritten by the associated body  $E$  of its defining equation  $X \stackrel{\text{df}}{=} E$ .

A dynamic context, here, is any  $\mathcal{C}(\bullet)$  such that, when R.TOYA expressions  $X$  and  $E$  are inserted into the place of the bullet,  $\mathcal{C}(X)$  and  $\mathcal{C}(E)$  yield dynamic expressions<sup>5</sup>. The relation of ‘rewriting’ is here formally denoted by  $\xrightarrow{\emptyset}$ , because in the (intended) Petri net semantics, this should be considered as an empty step, i.e., a non-move or a *non-change*<sup>6</sup>. In the first step of the derivation shown above, we have the context  $\overline{(\bullet)}$  with the bullet replaced by  $X$ , and the recursion principle allows us to rewrite  $X$  to the body  $E$ , that is, to  $a; X$ , and thus, in the given context, to  $\overline{a; X}$ .

Well, the same recursion principle may be applied at the end of the previous derivation, the context being  $a; \overline{(\bullet)}$ , and we will get as a result that another  $a$  can occur, thus:

$$a; \overline{X} \xrightarrow{\emptyset} a; \overline{a; X} \equiv a; (\overline{a}; X) \xrightarrow{a} a; (\underline{a}; X) \equiv a; (a; \overline{X}),$$

and the recursion principle can again be applied at this point. Clearly, the process may be repeated indefinitely, and so we may derive an infinite sequence of  $a$ -moves from  $\overline{X}$ .

<sup>5</sup> Dynamic R.TOYA expressions are defined similarly to dynamic B.TOYA expressions.

<sup>6</sup> This is to be distinguished from an *actual* move, however ‘silent’ it may be. For instance, a  $\tau$ -move would, as a step containing only one move, be written as  $\xrightarrow{\{\tau\}}$ .

Let us have a look at another, not so obvious, example:  $X \stackrel{\text{df}}{=} a + (X; a)$ . We claim that from  $\overline{X}$ , two  $a$ -moves can be made in succession. Here is a derivation:

$$\begin{aligned} \overline{X} &\xrightarrow{\emptyset} \overline{a + (X; a)} \equiv^2 a + (\overline{X}; a) \xrightarrow{\emptyset} a + (\overline{a + (X; a)}; a) \equiv a + ((\overline{a} + (X; a)); a) \\ &\xrightarrow{a} a + ((\underline{a} + (X; a)); a) \equiv^2 a + ((a + (X; a)); \overline{a}) \xrightarrow{a} a + ((a + (X; a)); \underline{a}) \end{aligned}$$

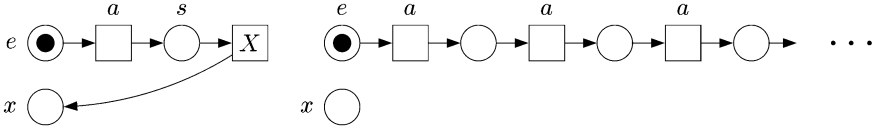
where  $\equiv^2$  abbreviates ‘two  $\equiv$ -steps’. Looking carefully at this derivation reveals that, (i), a sequence of  $a$ -moves of any arbitrary finite length can be derived from  $\overline{X}$ , but also: (ii), no *infinite* sequence of  $a$ -moves can be derived from  $\overline{X}$ , in contrast to the previous example. Property (i) can easily be seen by observing that the above derivation can be made longer by repeating some intermediate applications of the recursion principle, and (ii) is true because in order for any move to be made at all from  $\overline{X}$ , the *first*  $a$  in  $a + (X; a)$  has to be chosen first. Or, put differently, the above derivation cannot be extended at its tail, as the previous one could (we may continue to replace  $X$  in the dynamic expression  $a + ((a + (X; a)); \underline{a})$  by its body  $a + (X; a)$ , but we may never derive another move from it, because the underbar can never be ‘promoted’ to an overbar).

Now let us discuss a perhaps weird example:  $X \stackrel{\text{df}}{=} X$ . Surely, the recursion principle lets us rewrite  $\overline{X}$  indefinitely often into  $\overline{X}$ , but the expression does not change and no actual *move* will ever result from this, not even a tiny little  $\tau$ -move. This is particularly strange when we consider what could be the Petri net associated with such an  $X$ . Apparently, *any* Petri net satisfies an equation such as  $X = X$ , under any arbitrary (reasonable) notion of equality. However, consider any Petri net that has some transition that *can* occur. A move of such a transition cannot be derived from the above recursion principle alone, and thus, a reasonable *principle of economy*<sup>7</sup> allows us to exclude such nets from consideration as the semantics of  $X \stackrel{\text{df}}{=} X$ . This creates a huge simplification: we will admit as solutions of  $X \stackrel{\text{df}}{=} X$  only such Petri nets that have no activated transitions, and, by extension of the principle of economy, as solutions of a general equation  $X \stackrel{\text{df}}{=} E$  only those nets with ‘minimalistic’ behaviour<sup>8</sup>. Of course, a ‘most minimal’ Petri net with no activated transitions, is one which has no transitions at all. Nevertheless, our nets should have at least an  $e$ -labelled place and an  $x$ -labelled place, because otherwise they cannot easily be used as building blocks for larger nets. Thus, in the end, the Petri net corresponding to  $X \stackrel{\text{df}}{=} X$  has, by definition (and by the principle of economy, applied to places as well), exactly two places (one  $e$ -labelled and the other  $x$ -labelled), and no transition.

How, then, should the Petri nets associated with the other examples,  $X \stackrel{\text{df}}{=} a; X$  and  $X \stackrel{\text{df}}{=} a + (X; a)$ , look like? First of all, it is a good idea to construct nets for their bodies, with  $X$  treated as a simple transition. Figure 13 shows, on its left-hand side, the net corresponding to the body, i.e.  $a; X$ , of  $X \stackrel{\text{df}}{=} a; X$ .

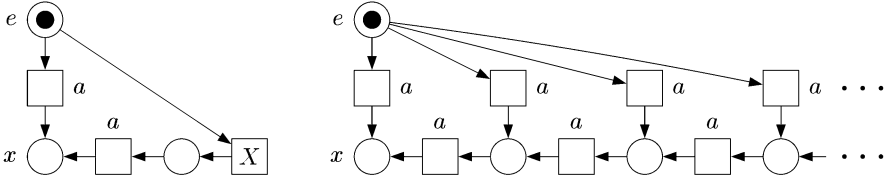
<sup>7</sup> Which states, more precisely, that the recursion principle is the *only* means, in addition to the rules for the other operators, of deriving moves for a recursive equation.

<sup>8</sup> This notion, which we slightly ironised here, can actually be made quite precise.



**Fig. 13.** Nets for  $X \stackrel{\text{df}}{=} a; X$ ;  $X$ : the body and solution.

As one can see, it has been constructed just in the same way as other previous examples. However, it is drawn in a slightly twisted way. Why? Well, look at what happens if we refine the  $X$  transition by the net shown on the right-hand side, identifying the place called  $s$  with the  $e$ -labelled place of the right-hand side, and also identifying the two  $x$ -labelled places: we will get back ‘the same’ net as already shown on the right-hand side. The refinement into  $X$  just serves as a prolongation of the net to the left, or, seen in another way, ‘pushes’ the chain of  $a$ -labelled transitions one position further to the right, without changing anything about the structure. Actually, it is possible (and done in [6]) to define the refinement operator in such a way that we may omit the apostrophes around ‘the same’ above: *exactly* the same net arises when the right-hand side net is refined into the  $X$  on the left-hand side: visually and formally, the net shown on the right-hand side is a *solution* of  $X \stackrel{\text{df}}{=} a; X$ , with  $\stackrel{\text{df}}{=}$  replaced by actual equality, and it is thus a *fixpoint* of the (Petri net) mapping  $f(X) = a; X$ .



**Fig. 14.** Nets for  $X \stackrel{\text{df}}{=} a + (X; a)$ ; the body and solution.

Exactly the same happens, in fact, in the other example, shown in Figure 14. Note that this net exhibits precisely the same behaviour as the one we derived from the expression  $X \stackrel{\text{df}}{=} a + (X; a)$  by SOS arguments: an arbitrarily long finite chain of  $a$ -moves is possible, but no infinite one. These two examples can be generalised: refinement works in all cases of recursion expressible in R.TOYA, and yields solutions with exactly the same step sequences as derivable from the SOS rules<sup>9</sup>. Most meaningful recursions will, in some way, be guarded, that is, be such that recursion variables do not occur immediately at the beginning of an execution of the right-hand side of a defining equation (thus,  $a; X$  is guarded while  $a + (X; a)$  is not). Without guards, however, more strange cases need to be

<sup>9</sup> Actually, one can prove strong (in the sense that not only moves, but steps, are considered) transition system isomorphism.

considered, for instance  $X \stackrel{\text{df}}{=} (a \parallel b) + X$ . A little reflection shows that it should have an uncountably large solution – but does it also have countable solutions? *With* guards, however, one will usually have the unique solution property (see, e.g., [3]), and this is also true in ‘minimal’ Petri net models.

The recursion principle is present, in one form or another, in most process algebras. For instance, basic CCS recursion is explained in [25] thus<sup>10</sup>:

The recursion rule for  $X \stackrel{\text{df}}{=} E$  says, informally, that any action which may be inferred for the body  $E$ , ‘unwound’ once (by substituting itself for its bound variable) may be inferred for  $X$  itself.

In [3], a model for a (guardedly) recursive equation is explicitly constructed by repeatedly substituting the body,  $E$ , of  $X \stackrel{\text{df}}{=} E$  into  $X$ ’s that (re)appear on the right-hand side, to get better and better approximations of the solution which is taken as a (projective) limit of these approximations.

## 8 Concluding Remarks

We have discussed some of the issues arising in giving Petri net semantics to process algebras, as well as some of the issues arising in process algebra in general, not specifically in the context of Petri nets. Why should one want to define a translation from process algebra to Petri nets? Generally speaking, this may be useful as soon as one wishes to attempt exploiting Petri-net based techniques and/or concepts for the analysis and/or the design of process-algebra based systems. In [24], a translation from a concurrent language (close to the ones in actual use) into basic CCS was given. The PEP system (Programming Environment based on Petri nets [31]) uses a similar translation, as well as a translation from concurrent programs (and inputs described in other concurrent systems notations) into M-nets ([23], a process algebra with a high-level Petri net semantics), and from there into low-level (basic) nets like the ones considered in this paper, and further into their unfoldings [22]. Travelling back and forth on this chain of translations, unfolding-based (and more generally, Petri-net based) verification methods (e.g., [13]), can be applied to concurrent programs.

Finally, we would like to discuss very briefly an approach which is directly opposite to that presented in this paper. Roughly speaking, one might be interested how can a process algebraic expression be derived from a given Petri net? A first point to note is that such an expression (if any) is not likely to be unique, and there may even be several systematic constructions which yield different results. For instance, in B.TOYA one may write two very dissimilar expressions

$$(a \parallel b); \tau; (a \parallel b) \quad \text{and} \quad ((a; (c; a)) \parallel (b; (\hat{c}; b))) \text{ sy } c \text{ rs } c$$

which have the same Petri net semantics. Another point to note is that a Petri net is not, in general, structured. Thus, outside information has to be used when

<sup>10</sup> Page 57. This is not a literal quote, but hopefully conveys the intended meaning.

searching for a good translation into a process algebra. Having said that, in special cases, systematic constructions can be given. For instance, if the Petri net is 1-bounded and covered by S-components, then, using COSY, a translation of such a net into a path expression (and further into an expression of CCS or of TCSP) may be devised. And, if a Petri net is such that every place has only one input arc, other systematic translations can also be found.

## Acknowledgment

The first author gratefully acknowledges financial support by the research project UK/EPSRC BESST – Behavioural Synthesis of Systems with Heterogenous Timing (grant GR/R16754).

## References

1. L. Aceto, W.J. Fokkink, C. Verhoef: Structured operational semantics. In [4], 197-292.
2. J.C.M. Baeten, C.A. Middelburg: Process algebra with timing: Real time and discrete time. In [4], 627-684.
3. J.C.M. Baeten, W.P. Weijland: **Process Algebra**. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.
4. J.A. Bergstra, A. Ponse, S.A. Smolka (eds): **Handbook of Process Algebra**. Elsevier Science B.V., Amsterdam, 2001.
5. E. Best: **Semantics of Sequential and Parallel Programs**. Prentice Hall, 1996.
6. E. Best, R. Devillers, M. Koutny: **Petri Net Algebra**. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 2001.
7. G. Boudol, I. Castellani: Flow models of distributed computations: Three equivalent semantics for CCS. *Information and Computation* 114(2), 247-314, 1994.
8. R.H. Campbell, A.N. Habermann: The specification of process synchronization by path expressions. *Symposium on Operating Systems*, 89-102, 1974.
9. I. Castellani: Process algebras with localities. In [4], 945-1045.
10. S. Christensen, Y. Hirshfeld, F. Moller: Bisimulation is decidable for basic parallel processes. *Proc. of CONCUR'93, Lecture Notes in Computer Science* 715, Springer-Verlag, 143-157, 1993.
11. P. Degano, R. De Nicola, U. Montanari: A partial ordering semantics for CCS. *Theoretical Computer Science* 75(3), 223-262, 1990.
12. E.W. Dijkstra: **A Discipline of Programming**. Prentice Hall, 1976.
13. J. Esparza: Model checking using net unfoldings. *Science of Computer Programming* 23, 151-195, 1994.
14. R. van Glabbeek: The linear time – branching time spectrum I. In [4], 3-99.
15. R. van Glabbeek, U. Goltz: Refinement of actions in causality based models. *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. Lecture Notes in Computer Science* 430, Springer-Verlag, 267-300, 1990.
16. R. van Glabbeek, F.W. Vaandrager: Petri net models for algebraic theories of concurrency. *Proc. of PARLE, Parallel Architectures and Languages, Vol. II, Lecture Notes in Computer Science* 259, Springer-Verlag, 224-242, 1987.

17. U. Goltz: **Über die Darstellung von CCS-Programmen durch Petrinetze**. Dissertation, Gesellschaft für Mathematik und Datenverarbeitung, 1988.
18. C.A.R. Hoare: Communicating sequential processes. *Comm. of the ACM* 21, 666-677, 1978.
19. C.A.R. Hoare: **Communicating Sequential Processes**. Prentice Hall, 1985.
20. R. Janicki and P.E. Lauer: **Specification and Analysis of Concurrent Systems – the COSY Approach**. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.
21. R. Keller: Formal verification of parallel programs. *Comm. of the ACM* 19, 371-384, 1976.
22. V. Khomenko, M. Koutny, W. Vogler: Canonical prefixes of Petri net unfoldings. *Acta Informatica* 40, 95-118, 2003.
23. H. Klaudel, F. Pommereau. M-nets: a survey. Manuscript, 2003 (submitted).
24. R. Milner: **A Calculus of Communicating Systems**. Lecture Notes in Computer Science 92, Springer-Verlag, 1980.
25. R. Milner: **Communication and Concurrency**. Prentice Hall, 1989.
26. E.R. Olderog: **Nets, Terms and Formulas**. Cambridge Tracts in Theoretical Computer Science 23, Cambridge University Press, 1991.
27. D. Park: Concurrency and automata on infinite sequences. *Proc. 5th GI Conference, Karlsruhe*, Lecture Notes in Computer Science 104, Springer-Verlag, 167-183, 1981.
28. G.D. Plotkin: A structural approach to operational semantics. Report FN-19, Computer Science Department, University of Aarhus, 1981.
29. L. Priese, H. Wimmel: **Petri-Netze**. Springer-Verlag, Theoretische Informatik, 2003.
30. P. Sewell: Nonaxiomatisability of equivalences over finite state processes. *Annals of Pure and Applied Logic*, 90(1-3), 163-191, 1997.
31. Ch. Stehno: Real-time systems design with PEP. *Proc. TACAS'02*, Lecture Notes in Computer Science 2280, Springer-Verlag, 476-480, 2002.
32. D. Taubner: **Finite Representation of CCS and TCSP Programs by Automata and Petri Nets**. Lecture Notes in Computer Science 369, Springer-Verlag, 1989.
33. G. Winskel: Petri nets, algebras, morphisms and compositionality. *Information and Control* 72, 197-238, 1987.