Petri Nets, Process Algebras and Concurrent Programming Languages*

Eike Best

Fachbereich Informatik, Carl-von-Ossietzky-Universität zu Oldenburg,
D-26111 Oldenburg, Germany, email: eike.best@informatik.uni-oldenburg.de

Raymond Devillers

Département d'Informatique, Université Libre de Bruxelles,
B-1050 Bruxelles, Belgium, email: rdevil@ulb.ac.be

Maciej Koutny

Department of Computing Science, University of Newcastle upon Type,
Newcastle upon Tyne, United Kingdom, email: maciej.koutny@newcastle.ac.uk

Abstract

This paper discusses issues that arise when process algebras and Petri nets are linked; in particular, operators, compositionality, recursion, refinement and equivalences. It uses the box algebra in order to show how Petri nets can be manipulated algebraically. Also, the paper shows how process algebras such as CCS, COSY and CSP can be treated in the same way, how Petri net semantics of concurrent programming languages can be given, and how Petri net methods can be applied to the verification of concurrent algorithms.

1 Introduction

One of the main aims of the Petri net theory is to model concurrent systems and to allow to reason about them formally. The question of realising such systems is not primarily addressed, although it is clearly desirable that a system modelled by a net should also be realisable. One of the main aims of concurrent programming languages is to express parallel programs and distributed algorithms, and thus to construct concurrent systems. The question of reasoning about them remains in the background, although it is clearly desirable that the properties of such systems should be amenable to rigorous analysis. Process algebras are akin to Petri nets in that they provide formal means of reasoning

*This work has been done in cooperation with other people, in particular **Javier Esparza, Jon G. Hall and Richard P. Hopkins**. It has been supported by the Esprit Basic Research Project 3148 DEMON (Design Methods Based on Nets) and Working Group 6067 CALIBAN (Causal Calculi Based on Nets).

about concurrent systems, and also akin to concurrent programming languages in that they provide syntactic means for constructing concurrent systems.

The authors of this tutorial paper are working on the hypothesis that there is some benefit to be gained from a sustained attempt to discover just how closely these different means of describing and analysing concurrent systems are linked. Thus, it was not our plan to write three isolated or loosely related manuscripts combined into a single article. Nevertheless, the three subthemes of this paper are, by themselves, so large and well investigated that any attempt to cover the relevant material exhaustively would result in an utter failure.

Therefore, we have decided to concentrate mainly on one class of objects for each of the themes: one class of nets, *safe place/transition nets*, one process algebra, *PBC (Petri Box Calculus)*, and one class of concurrent programs, *shared variable programs*. We intend to focus on what could be seen as the fundamental similarities and differences between them. While discussing these, we will gain insight which might be used in order to describe similarities and differences between other classes of nets, other process algebras, and other types of programming languages. At the same time, we aim to emphasise some of the central concepts relating to each of the three classes of concurrent systems description techniques.

The paper is structured as follows. In section 2, we discuss informally the syntax and semantics of process algebras. In particular, we focus on possible ways of defining 'parallel composition', which is a useful (and widely used) operator for the description of parallel systems, and on iteration (or recursion), which is indispensable in order to describe repetitive behaviour. We concentrate on motivating the particular operators that can be found in PBC, but we do not claim that these are the only possible choices and, to reinforce this view, we discuss alternatives. Later, in section 5, we also show that there is a way of encapsulating the various PBC operators by viewing them as incarnations of a more abstract meta-operator.

In section 3, we discuss two of the basic ingredients of process algebra theory: operational semantics and equivalence notions. Again, we discuss these notions specifically for PBC. However, if readers are interested in other process algebras (and we will point out some books and other readily available material for them), they may well still benefit from this discussion, because operational semantics and behavioural equivalence appear there too, in one form or another. One of the points we will also investigate is the relationship between structure and behaviour; how these two notions are sharply distinguished in Petri net theory, but less so in process algebras; and how this discrepancy can be resolved by means of structural identities. Another notion we discuss is that of a congruence relation between process algebraic expressions.

After reading section 3, the reader will be equipped with a syntax and an operational semantics for PBC. From section 4 onwards, we turn to the Petri net semantics of PBC – again, in lieu of other process algebras. That is, PBC with the same syntax as in the previous section will be provided with a translation into the chosen class of Petri nets. We will formulate requirements for this translation (mainly that it be compositional), and show that generalised transition refinement is adequate for this purpose. The second part of section 4 provides an explicit translation from PBC expressions to nets.

At this point, the syntax and two semantics of PBC are available, namely the operational semantics and Petri net semantics (the latter is sometimes called a denotational semantics). Section 5 develops the theory that is necessary for showing that the two are actually equivalent. In the course of developing this (incidentally, rather strong) equivalence result, we will find it convenient to generalise the concept of process algebra syntax in such a way that not only the expressions of the algebra, but also its operators, are describable by Petri nets from the chosen class. In other words, sequential composition, choice composition, parallel composition, and a whole (infinite) class of other operators, will be described by certain characteristic Petri nets, and for all of them, the equivalence result can be shown to hold. Moreover, it turns out that all the rules of operational semantics can be viewed as instantiations of a general meta-rule.

Having thus obtained a very tight relationship between the process algebra and the class of Petri nets under consideration, in section 6 we turn to the realm of concurrent programming languages. The language we consider there is sufficiently simple so that its semantics (in terms of the process algebra, and hence also the chosen class of nets) can be defined in a few lines, yet it is also sufficiently expressive to allow some basic nontrivial parallel algorithms to be formulated. Beside the semantics of this language, we describe three mutual exclusion algorithms, and show that Petri net methods can be used effectively (and also automatically, using a computer-aided verification tool) to yield the correctness proofs of such algorithms.

To keep the paper (approximately) within the page limits given to us, some material has had to be omitted. First of all, we have omitted all proofs of the claims we make. The reader may turn to [28], and in particular to [8], in order to find the proofs. Secondly, we have omitted the treatment of recursion, as far as its Petri net semantics is concerned; this relies on fixpoint theory and is more complicated than its operational semantics, which will be described in section 3.2.7. Readers interested in the Petri net semantics of recursion will find it both in a tutorial paper [10] and in a paper describing the full theory [28]. Thirdly, we have neglected a whole body of net theoretical results pertaining to the structural well-behavedness of nets which correspond to process algebraic expressions. The interested reader can find such theory in [16].

2 The Basic Petri Box Calculus

We discuss a process algebra whose name – Petri Box Calculus (PBC) – arises from its original [6, 7] Petri net semantics. The PBC combines a number of features taken from other existing process algebras, notably COSY [26], CCS [31, 32], SCCS [32], (T)CSP [25] and ACP [1]. But there are also some differences with each of these process algebras since PBC has been designed with two specific objectives in mind: to support a compositional Petri net semantics, together with an equivalent – more syntax-oriented – structured operational semantics (SOS) [39], and to provide a sound, and as flexible as possible, basis for a compositional semantics of high level concurrent specification and programming languages with full data and control features.

The aim of this section is to acquaint the reader with the main points and topics that would need to be discussed when such a process algebra is constructed. Since Milner's CCS can be viewed as directly inspiring the design of PBC, we start with it.

2.1 Informal introduction to CCS

A process algebra is usually constructed from a set of basic processes and a set of operators, each operator having a fixed arity indicating the number of its operands. For instance, the standard basic CCS corresponds to the following syntax:

$$E_{ccs} ::= \mathsf{nil} \mid a.E_{ccs} \mid \tau.E_{ccs} \mid E_{ccs}{+}E_{ccs} \mid E_{ccs} \,|\, E_{ccs} \mid E_{ccs}[f] \mid E_{ccs}\backslash a \mid X. \tag{1}$$

This syntax contains a single basic process nil, an infinite family of unary prefix operators parameterised by action names a, together with the silent prefixing unary operator τ, two binary infix operators (the choice, $+$, and the composition, $|$), and two infinite families of unary postfix operators (the restrictions $\backslash a$, also parameterised by action names, and the relabellings $[f]$, parameterised by functions f acting on action names). Variables X may also be considered as basic processes, but they are associated with defining expressions of the form $X \overset{\mathrm{df}}{=} E_{ccs}$. In this way, variables support different levels of abstraction within a CCS specification, as well as recursion.

Every process has an associated set of behaviours. It is one of the objectives of formal semantics to make this notion precise. Here, however, we will only describe informally the semantics of CCS expressions.

The basic process nil 'does nothing'. $a.E_{ccs}$ does a and thereafter behaves like E_{ccs}. $\tau.E_{ccs}$ does τ and thereafter behaves like E_{ccs}. The difference is that τ is an internal ('silent') action while a is not. $E_{ccs} + F_{ccs}$ behaves either like E_{ccs} or like F_{ccs}. $E_{ccs} \,|\, F_{ccs}$ behaves like both E_{ccs} and F_{ccs}, together with some further (synchronisation) activities. $E_{ccs}[f]$ behaves like E_{ccs}, except that function f is applied to all actions. $E_{ccs}\backslash a$ behaves like E_{ccs}, except that the action a may not be executed. Finally, X behaves like the process E_{ccs} in its defining equation $X \overset{\mathrm{df}}{=} E_{ccs}$ (and the truly recursive case arises when X re-occurs, directly or indirectly, within E_{ccs}). Consider, for instance, the CCS expression $E_0 = a.(b.\mathsf{nil})$ which can 'make an a-move', thereafter 'make a b-move', and then terminate. This is formalised in CCS as the sequence

$$\underbrace{a.(b.\mathsf{nil})}_{E_0} \xrightarrow{a} \underbrace{b.\mathsf{nil}}_{E_1} \xrightarrow{b} \underbrace{\mathsf{nil}}_{E_2}.$$

The example exhibits an important feature of the CCS treatment of expressions and their behaviours. E_0, E_1 and E_2 are all different CCS expressions with different structure. In other words, the original expression has changed its structure through a behaviour. In CCS, and indeed several other process algebras, the structure and behaviour of process expressions are intertwined in this way.

2.2 Informal introduction to Petri nets

The graph of a Petri net N describes the structure of a system the net is supposed to represent. The *behaviour* of this system is defined with respect to a given starting marking (state) of the graph, which is usually called the *initial marking*. A *marking* M of N is a function from the set of places of N into the set of natural numbers $\mathbf{N} = \{0, 1, 2, \ldots\}$. A transition t is called *enabled by M* if all input places of t carry one or more token. If an enabled transition occurs, then this is tantamount to the following change of marking: a token is subtracted from each input place of t; a token is added to each output place of t; and no other places are affected. We use the notation $M \xrightarrow{t} M'$

(or $(N,M) \xrightarrow{t} (N,M')$, to emphasize N) in order to express that M enables t and M' arises out of M according to the rule just described; M' is then said to be reachable from M in one step, and the reflexive and transitive closure of this relation gives the general reachability relation. A marking is called *safe* if it returns 0 or 1 for every place, and a marked net is called *safe* if every marking reachable from the initial marking is safe.

Figure 1 shows an unmarked Petri net, N, and two marked Petri nets, (N,M_0) and (N,M_1). Note that the underlying net itself is the same in all cases, only the markings are different. Note also that the unmarked net on the left could as well be interpreted as a marked net, namely as the net which has the 'empty' marking assigning the number 0 to all places.

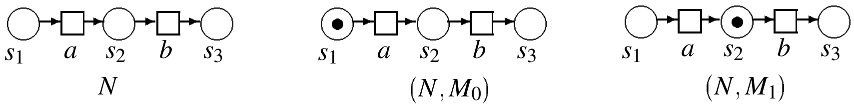


Figure 1: A Petri net with an empty and two nonempty markings.

Formally, the net N of Figure 1 would be described as a triple (S,T,W) where S is the set of its places, $S=\{s_1,s_2,s_3\}$, T is the set of its transitions, $T=\{a,b\}$, and W assigns a number (in this case, only 0 or 1) to pairs (s,t) and (t,s), where $s\in S$ and $t\in T$, depending on whether or not an arrow leads from s to t or from t to s, respectively. We have, for instance, $W(s_1,a)=1$, $W(s_1,b)=0$, $W(a,s_1)=0$, and so on (1 stands for 'arrow', 0 stands for 'no arrow'). The marking M_0 shown in the middle of the figure would be described as a function with $M_0(s_1)=1$, $M_0(s_2)=0$ and $M_0(s_3)=0$, and the marking M_1 shown on the right of the figure as a function with $M_1(s_1)=0$, $M_1(s_2)=1$ and $M_1(s_3)=0$. Note that M_0 enables transition a. This transition may, hence, occur, and if it does, then the resulting marking is M_1. Thus, $(N,M_0) \xrightarrow{a} (N,M_1)$.

Clearly, N describes a sequence between the two transitions a and b, and it is therefore related to the CCS expression $a.(b.\mathsf{nil})$ discussed earlier. But which of the three nets should correspond to the semantics of $a.(b.\mathsf{nil})$? The answer is clear: the middle one. And the reason is that it is only this net, rather than any of the other two, that has the same *behaviour* as $a.(b.\mathsf{nil})$. The net N by itself has no behaviour, and in the marked net (N,M_1), b can occur as the first action, but not a as required by $a.(b.\mathsf{nil})$. Note what happens when the action a is executed in either model. In terms of CCS, the expression $a.(b.\mathsf{nil})$ is transformed into the expression $b.\mathsf{nil}$. In terms of nets, the marked net (N,M_0) is transformed into the marked net (N,M_1), with the same underlying net N. If we were to define a Petri net corresponding to the expression $b.\mathsf{nil}$, however, then we would not, of course, think of taking the net (N,M_1); rather, the obvious choice would be a marked net with two places and one transition b, i.e., (N,M_1) after deleting s_1 and a.

This example shows that the ways of generating behaviour in net theory and in CCS do not directly correspond to each other. In the latter, the structure of an expression may change, while in the former, only markings change but the structure of a net remains the same through any behaviours. The difference is a fundamental one, and stems from the underlying ideas of modelling systems. In net theory, when transition a has occurred, we may still recover its presence from the static structure of the system, even though it may never again be executed; there is thus a sharp distinction between the static and the dynamic aspects of a system. In CCS, when action a has occurred, as in the above context, then we may safely forget about it, precisely because it may never be executed, and hence we may safely change the structure of the expression; there is thus a very close relation between the static and dynamic aspects of expressions.

2.3 Structure and behaviour of PBC expressions

Because, in PBC, we wish to have a smooth Petri net semantics, we will devise a behavioural semantics for it which respects the division between static and dynamic aspects that can be found in Petri net theory. The basic idea will be to introduce into the syntax of expressions something that models markings. Thus, we will define expressions 'without markings' (called 'static expressions') and expressions 'with markings' (called 'dynamic expressions'): static expressions correspond to unmarked nets, while dynamic expressions correspond to marked nets. Unless stated otherwise, we will always use E and F to denote static expressions, and G and H to denote dynamic expressions.

As an example, consider the static PBC expression $E = a;b$. It corresponds to the unmarked net N on the left-hand side of Figure 1. Making expressions dynamic consists of overbarring or underbarring (parts of) them. For instance, consider the dynamic PBC expression $\overline{E} = \overline{a;b}$. By definition, it corresponds to the same net as E, but with a marking that lets E be executed from its start, i.e., to the 'initial marking' of E. The net corresponding to $\overline{E}$ is the marked net (N, M_0) shown in the middle of figure 1. Next consider the – syntactically legal and meaningful – dynamic PBC expression $G = \underline{a};b$. By definition, it corresponds to the same underlying expression E, but in a state in which its first action a has just been executed, i.e., it shows the instant at which the final state of a has just been produced. Thus, G corresponds to the marked net (N, M_1) shown on the right-hand side of Figure 1.

This syntactic device of overbarring and underbarring introduces what could, at first glance, be seen as a difficulty. For consider the same expression E in a state in which its second part, b, is just about to be executed: $G' = a;\overline{b}$, which is again syntactically legal. Which marked net should this dynamic expression correspond to? There is only one possible reasonable answer to this question, namely the right-hand side net, (N, M_1), of Figure 1. Thus, in general, we may have a many-to-one relationship between dynamic expressions and marked nets. Let us use the symbol $\equiv$ in order to relate dynamic expressions which are not necessarily syntactically equal, but are in a relationship such as G and G' in the example. Then $G \equiv G'$ can be viewed as expressing that 'the state in which the first component of a sequence is terminated is the same as the state in which the second component may begin to be executed'. Note that $\overline{E} = \overline{a;b}$ also has a dynamic expression which is equivalent but not syntactically equal, namely $H = \overline{a};b$. $\overline{E} \equiv H$ can be viewed intuitively as expressing that 'the initial state of a sequence equals the initial state of its first component'.

We do not see this many-to-one relationship as merely incidental, to be overcome, perhaps, by a better syntactic device than that of overbarring and underbarring subexpressions. Instead, we view it as evidence of a fundamental difference between Petri nets and process algebras: expressions of the latter come with in-built structure, while Petri nets do not. For instance, $E = a;b$ is immediately recognised as a sequence of a and b. In the example of Figure 1, one may also recognise the structure to be a sequence. However, this is due to the simplicity of the net. For an arbitrary bipartite graph it is far from obvious how it may be seen as constructed from smaller parts, while it is always clear, for any well-formed expression in any process algebra, how it is made up from subexpressions. The fact that an expression of a process algebra always reveals its syntactic structure, has to be viewed as a great advantage which has some highly desirable consequences. For example, it is often possible to argue 'by syntactic induction'; one may build proof systems 'by syntactic definition' around such an algebra; and the availability of operators for the modular construction of systems is usually appreciated by practitioners. On the other hand, one of the disadvantages of such a syntactic view is that some effort has to be invested into the definition of the behaviour of expressions. While this can usually be done inductively, it is still necessary to go through all operators one by one and to define their semantics individually. Sometimes, this leads to ad-hoc (and very disparate) definitions. A non-structured model such a Petri nets has a clear advantage in this respect; its behaviour (with respect to a marking) is defined by a single rule, namely the transition rule, which covers all cases.

In this spectrum, the PBC attempts to cover a middle position. PBC expressions, whether they are static or dynamic, still come with an in-built structure. This has the desirable consequences mentioned above (such as the potential for inductive reasoning). On the other hand, the behavioural rules of (dynamic) PBC expressions are, in fact, the Petri net transition rule in disguise; we will make this point clear later. The necessity to consider equivalences such as $\equiv$ can be viewed as being the price that has to be paid for being able to combine advantages of both Petri nets and process algebras in the context presently considered. However, if it is indeed a price, then we are willing to pay it, because when the analysis of the $\equiv$ equivalence is carried through in detail (and we will do this in section 5), it turns out that one gains sufficient conditions for nets to be operators. This is a good result to have, of course, because it separates the desirable objects (or, at least, a large class of desirable objects) from the rest. Some such operators are discussed in the next sections.

2.4 Sequential composition

Instead of, as in CCS, expressing sequential behaviour by so-called 'prefixing', $a.F$, we may also use, as in PBC, a true sequential operator, $E;F$, meaning that E is to be executed before F. The main difference is that in CCS, E, the first part of a sequential composition, may only have the special shape $E = a$. The full sequence can be simulated in CCS by a combination of prefixing, parallel operator and synchronisation. One reason for allowing full sequential composition is that in a majority of imperative languages, this is one of the basic operations (and it is usually denoted by the semicolon) allowing one to put in sequence any two subprograms. A second reason is that the semantics of the full sequential composition is no more complicated, in PBC, than the semantics of the prefixing.

The nil process is no longer necessary in PBC, and thus it will be dropped. To see this, we compare the CCS way of deriving the behaviour of the CCS expression $a.(b.\mathsf{nil})$ and the PBC way of deriving the behaviour of the dynamic PBC expression $\overline{a;b}$:

$$\text{In CCS:}\quad a.(b.\mathsf{nil}) \xrightarrow{a} b.\mathsf{nil} \xrightarrow{b} \mathsf{nil}.$$
$$\text{In PBC:}\quad \overline{a;b} \equiv \overline{a};b \xrightarrow{a} \underline{a};b \equiv a;\overline{b} \xrightarrow{b} a;\underline{b} \equiv \underline{a;b}.$$

Note that in both cases, the occurrence sequence ab is derived, but that the PBC way of deriving this behaviour is more symmetric while the CCS way is shorter. In CCS, the fact that 'execution has ended' is expressed by the derivation of nil, while in the PBC it is expressed by the derivation of an expression which is completely underbarred, such as $\underline{a;b}$. In general, if E is an arbitrary static expression, and if we call $\overline{E}$ 'the initial state of E', then we might as well – and will, in fact – call $\underline{E}$ 'the final (or terminal) state of E'. The reader must be warned, however, that this terminology is slightly deceptive, for there may be expressions E for which no behaviour leads from $\overline{E}$ to $\underline{E}$. This may happen either if from $\overline{E}$ a deadlock can be entered, or if from $\overline{E}$ an infinite loop can be entered, from which it is impossible to escape. Thus, when $\underline{E}$ is called a 'final state', this does not imply that this state is reachable; in the same way nil may be unreachable in CCS.

The raison d'être for the CCS process constant nil can be thus appreciated: it provides a syntactic way of describing final states. Given the CCS way of describing sequential behaviour by removing the prefixes of an expression as they get executed, there has to be a syntactic 'something' to describe the rest of the expression, once the last action is executed. From the above discussion, it may already be guessed that the Petri net semantics of nil ('what is the net – if any – corresponding to nil'?) is not obvious. In the literature, discussion about this question can be found [21, 43].

2.5 Synchronisation

As CCS, the PBC model is based upon the idea that in a distributed environment all complex interactions can be decomposed into primitive binary interactions. Let us first recall the meaning of the CCS synchronisation. Suppose that two people A and B wish to communicate by exchanging a handshake. In CCS, one could represent the action of 'A extending his hand towards B' by some symbol, say a; the action of 'B extending her hand towards A' by another symbol denoted by $\widehat{a}$ (the 'conjugate'[1] of a); and the fact that A and B are standing face to face by the symbol $\mid$, yielding the CCS expression $a \mid \widehat{a}$ (more exactly $(a.\mathsf{nil}) \mid (\widehat{a}.\mathsf{nil})$, but we may ignore nil here for simplicity).

The CCS expression a has only one possible activity: an execution of a, which represents the fact that A extends his hand towards B. Similarly, the CCS expression $\widehat{a}$ has only one possible activity: B extends her hand towards A. The expression $a \mid \widehat{a}$, however, by CCS definition, has the following possible activities: (i) A may extend his hand and withdraw it, leaving B with the same possibility; (ii) B may extend her hand and withdraw it, then A does the same; and (iii) both of them extend their hands which results in an actual handshake. More formally, using existing Petri net semantics of CCS expressions (e.g., in [21, 43]), these three possible activities could be represented in Petri net terms as shown in figure 2. The actual handshake – that is, the transition labelled

[1] In CCS, conjugation is usually denoted by overbarring; we use hatting instead, because overbarring is reserved for dynamic expressions, as explained in the previous sections.

τ – removes the possibility of further shaking of hands between A and B. The special CCS symbol τ is interpreted as an internal or silent action. In terms of the example, it indicates that the actual handshake is 'known' only to the two people involved, namely A and B, and has no external repercussions.

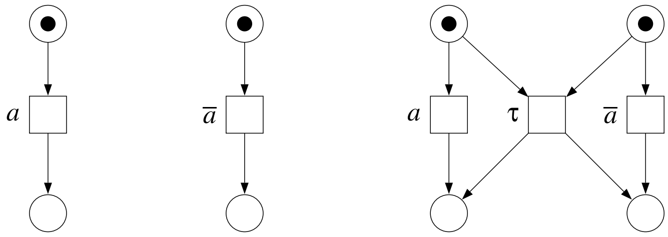


Figure 2: Activities of a (left), $\widehat{a}$ (middle) and $a \,|\, \widehat{a}$ (right).

Let us clarify at this point some terminology that is borrowed from net theory and is not standard in CCS. We will call the carriers of activity *transitions* and the entities such as a and $\widehat{a}$ in the above example *actions*. Thus, the CCS expression a has one transition which also corresponds to the action a. Similarly, the expression $\widehat{a}$ has one transition corresponding to $\widehat{a}$. However, the expression $a \,|\, \widehat{a}$ has three transitions although only two actions occur in it. It may be said that executing a transition corresponds to an actual activity, normally changing the state of the system, while an action is the way in which this transition is perceived from outside, i.e., the interpretation of the activity; in the Petri net theory this is often formalised through a labelling of transitions by actions. A similar observation can be made for CCS expressions such as $a.(a.\mathsf{nil})$. In the Petri net of this expression, there would be two transitions, even though the expression contains only one single action (which can be executed twice).

Now consider the question of how one could describe a handshake between three people A, B and C. In terms of Petri nets, such an activity could be modelled, quite similarly as before, by a three-way transition such as H in figure 3. A moment's reflection shows that a 3-way synchronisation transition like this cannot directly be modelled using the 2-way handshake mechanism of CCS; the reason is precisely that τ cannot be synchronised with any other action (there is nothing like $\widehat{\tau}$ in CCS). While one might try to simulate the 3-way-handshake using a series of binary handshakes, PBC proposes instead to extend the CCS framework so that a 3-way (and, in general, n-way) synchronisation could be expressed. One possibility, which is akin to the ACP approach [1], might be to generalise the conjugation operation to a function or a relation that allows the grouping of more than just two actions; however, PBC takes a different approach, which is still based on considering primitive synchronisations to be binary in nature. This works through the usage of structured actions, which we will describe next. Although one might think that 3-way handshakes occur rarely in practice, we will argue later that n-way synchronisations make perfect sense in the modelling of programming languages, as does the particular way of describing them in PBC.

To understand the PBC way of describing multiway synchronisations, it is perhaps easiest to interpret a CCS pair of conjugate actions $(a, \widehat{a})$ in the original way [31] as denoting

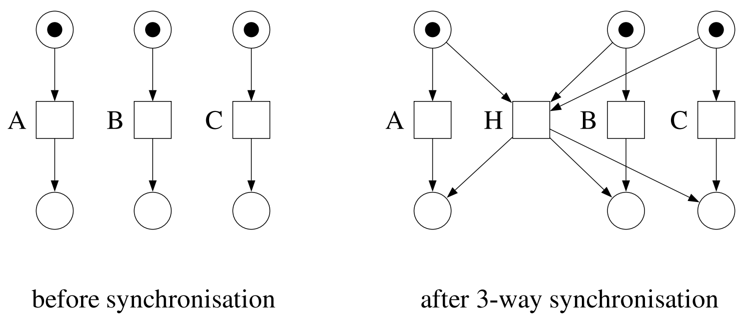


Figure 3: Handshake between three people.

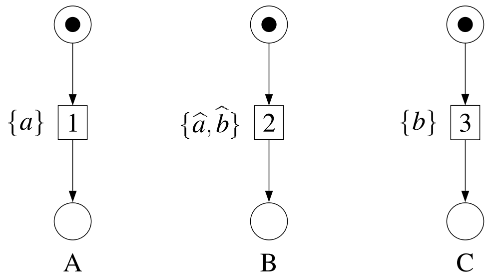
two links between two agents that fit together. In this view, a denotes a communication capability in search of a matching $\widehat{a}$ and, similarly, $\widehat{a}$ denotes a communication capability in search of a matching a. Once the communication takes place, i.e., a pair of a and $\widehat{a}$ is matched, this becomes a private synchronisation link between two agents with no further externally visible communication capability. The PBC model extends this idea in the sense that the result of a synchronisation does not have to be completely internal or 'silent', but instead, may contain further communication capabilities that may be linked with other activities. This generalisation is achieved by allowing a transition to correspond to a whole (finite) set of communication capabilities, such as for instance, $\emptyset$, $\{a\}$, $\{\widehat{a}\}$, $\{a,b\}$, $\{\widehat{a},b\}$, or even $\{a,\widehat{a},b\}$. For instance, an activity $\{\widehat{a}\}$ may be synchronised with activity $\{a,b\}$ using the conjugate pair $(a,\widehat{a})$. This results in a synchronised activity which is not silent but rather, still has the communication capability $\{b\}$, as the pair $(a,\widehat{a})$ is internalised but b remains untouched; formally: $(\{\widehat{a}\}\cup\{a,b\})\setminus\{\widehat{a},a\} = \{b\}$. For instance, figure 4 describes a 3-way handshake in terms of the (Petri net view of the) PBC (all possible synchronisations are shown there).

In the figure, a denotes 'A shakes hand with B', $\widehat{a}$ denotes 'B shakes hand with A', $\widehat{b}$ denotes 'B shakes hand with C' and b denotes 'C shakes hand with B'. Then, by the fact that B performs $\widehat{a}$ and $\widehat{b}$ simultaneously (expressed by the fact that $\{\widehat{a},\widehat{b}\}$ occurs as the label of a single transition), the resulting activity 123 (in figure 4) describes a simultaneous handshake between all three people. Transition 12 describes the handshake only between A and B (with label $\{\widehat{b}\}$ and capability to link with C) and transition 23 describes the handshake only between B and C (with label $\{\widehat{a}\}$ and capability to link with A). Transition 123 can also be thought of either as a 2-way synchronisation between 1 and 23, or as a 2-way synchronisation between 12 and 3.

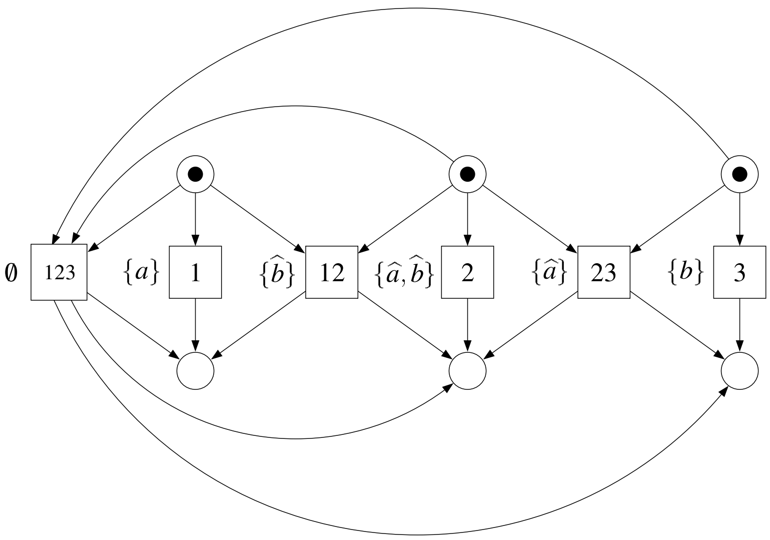
At this point, we need some basic definitions concerning multisets. A multiset over a set A is a function $\mathsf{m} : A \to \mathbf{N}$. A multiset m is finite if its support, i.e., $\{a{\in}A \mid \mathsf{m}(a){>}0\}$, is finite. For a finite multiset m, its cardinality is defined as the number $|\mathsf{m}| = \sum_{a\in A} \mathsf{m}(a)$. The operations of sum and (nonnegative) difference on multisets over A are defined as follows: for $a{\in}A$,

$$\begin{array}{lrcl} \text{Sum:} & (\mathsf{m}_1{+}\mathsf{m}_2)(a) & = & \mathsf{m}_1(a){+}\mathsf{m}_2(a) \\ \text{Difference:} & (\mathsf{m}_1{-}\mathsf{m}_2)(a) & = & \max(\mathsf{m}_1(a){-}\mathsf{m}_2(a),0). \end{array}$$

Note that if m_1 and m_2 are finite then so are $\mathsf{m}_1{+}\mathsf{m}_2$ and $\mathsf{m}_1{-}\mathsf{m}_2$. Multiset inclusion,



before synchronisation



after synchronisation

Figure 4: Three-way handshake in terms of PBC.

$m_1 \subseteq m_2$, is defined by $m_1(a) \leq m_2(a)$, for all $a \in A$. In the examples, we will write, e.g., $\{a,a,b\}$ for the multiset defined by $m(a) = 2, m(b) = 1$, and $m(c) = 0$ for all $c \in A \setminus \{a,b\}$. The set of finite multisets over A is denoted by $\mathsf{mult}(A)$.

Returning to the main discussion, let us consider the two PBC expressions $\{a,b\}$ and $\{\widehat{a},b\}$. According to the PBC approach, two transitions, say 1 and 2, corresponding, respectively, to these expressions can be synchronised using the conjugate pair $(a,\widehat{a})$. However, what synchronisation capability should the resulting transition 12 still possess? There are only two meaningful answers: either the set $\{b\}$ or the multiset $\{b,b\}$. The PBC model chooses the second alternative, and the next example shows the reason why. Consider a 'system with four people', $\{a,b\}$, $\{\widehat{a},b\}$, $\{\widehat{b}\}$ and $\{\widehat{b}\}$, and four transitions, say 1, 2, 3 and 4, corresponding to them. According to the rules of the game, transition 1 can be synchronised with transition 3 over $(b,\widehat{b})$, yielding a transition 13 with capability $\{a\}$. Similarly, 2 can be synchronised with 4 yielding a transition 24 with capability $\{\widehat{a}\}$. These two transitions can be further synchronised using $(a,\widehat{a})$ yielding a transition $(13)(24)$ with capability $\emptyset$, that is, a silent transition. No multisets are involved. Now suppose that we use first $(a,\widehat{a})$ in order to synchronise. Then transitions 1 and 2 can be synchronised to yield a transition 12. If this transition has only the capability $\{b\}$ then no further synchronisation involving $(b,\widehat{b})$-pairs will yield the four-way synchronisation obtained previously. If, however, the transition 12 has the capability $\{b,b\}$ then one of these b's can be used to synchronise with transition 3, the other to synchronise with transition 4, and the four-way silent synchronisation can be obtained in one of the following ways as well: $((12)3)4$ or $((12)4)3$, where the bracketing delineates (as it did previously) the individual binary CCS-like synchronisations. Since it is highly desirable that the order of the synchronisations is irrelevant, this explains why it is preferable to use multisets of communication capabilities instead of simple sets.

To summarise, PBC assumes given a set $\mathsf{A}_{\mathsf{PBC}}$ of primitive actions or *action particles*. These will be ranged over by $a,b,\ldots$, as in the above examples. Moreover, it is assumed that on $\mathsf{A}_{\mathsf{PBC}}$ there is defined a conjugation function $\widehat{\ }: \mathsf{A}_{\mathsf{PBC}} \to \mathsf{A}_{\mathsf{PBC}}$ with the properties of *involution* ($\widehat{\widehat{a}} = a$) and *discrimination* ($\widehat{a} \neq a$). This is the same basic setup as in CCS. However, unlike CCS, PBC allows as the label of a transition any finite multiset over $\mathsf{A}_{\mathsf{PBC}}$. That is, the set of *labels* (or, for the more process algebraically inclined, *elementary expressions*, or, to emphasize the fact that more than one action may be combined in a single transition, the set of *multiactions*) is defined as $\mathsf{Lab}_{\mathsf{PBC}} = \mathsf{mult}(\mathsf{A}_{\mathsf{PBC}})$.

This specialises to the basic CCS framework as follows: the CCS expressions a and $\widehat{a}$ correspond to the PBC expressions $\{a\}$ and $\{\widehat{a}\}$, respectively, and the CCS expression τ corresponds to the PBC expression $\emptyset$. No PBC action expression α with $|\alpha| > 1$ has a direct CCS equivalent.

2.6 Separating synchronisation from composition

Some of the CCS operators (nil, $+$, $a.$ and $\tau.$, often called 'dynamic connectives' in CCS terminology — but we will rename them as 'control (flow) connectives', in order to avoid confusion with dynamic PBC expressions) essentially act on the way the actions of the various components are organised in time with respect to each other, without depending on the shape of the components, i.e., they are only concerned with the control flow. Other ones ($\backslash a$ and $[f]$, often called 'static connectives' in CCS terminology — but we will again rename them as '(communication) interface connectives') do depend on the identity of the actions performed by the argument (modifying or selectively removing them). The CCS composition operator $\mid$ is special because it both performs a parallel composition of its arguments (control flow aspect) and adds synchronisations of conjugate pairs. While this is not necessarily harmful, it is valid to ask whether it would perhaps be semantically simpler to separate the control flow and the interface aspects of the last operator.

Another observation is that CCS parallel composition performs the synchronisation for all possible conjugate pairs. For instance, the CCS expression $a.(a.(b.\mathsf{nil}))\mid\widehat{a}.(\widehat{a}.(\widehat{b}.\mathsf{nil}))$ creates five synchronisations (three of them being actually executable); the syntax of CCS does not allow one to express synchronisation using only the pair $(b,\widehat{b})$ but not the pair $(a,\widehat{a})$. For reasons that will become clear in section 2.8 and later in section 6, it may be desirable to be able to say precisely which action names are being used for synchronisation. An obvious refinement of the CCS operator would be to allow selective parallel compositions as in TCSP [25]: the operator $\mid_a$ would denote that only $(a,\widehat{a})$ pairs may be used for synchronisation. However, in this case it is difficult to obtain some desirable algebraic laws such as associativity of parallel composition. Consider, for instance, $(\{b\}\mid_a\{a\})\mid_b\{\widehat{b}\}$ and $\{b\}\mid_a(\{a\}\mid_b\{\widehat{b}\})$. The first expression specifies a silent synchronisation between the first and the third transitions while the second expression specifies no such synchronisation. While it is of course possible to live with a restricted form of associativity, PBC proposes a different approach in order to obtain useful associativity properties. The approach consists of divorcing the synchronisation operator altogether from parallel composition and regarding synchronisation as a unary communication instead of binary control flow operator, denoted by $E\ \mathsf{sy}\ a$ where $a\in\mathsf{A}_{\mathsf{PBC}}$. What is gained by adopting this point of view is a reduction in the number of operators (otherwise $\mid_a$ and $\mid_b$ would have to be regarded as different), a general set of rules and a simple parallel operator. The non-associativity expected for expressions of the kind considered above then arises from the non-distributivity of the synchronisation operator over the parallel composition.

Considering synchronisation as a unary operator which is separate from the parallel composition makes it very similar to the restriction operator $\backslash$ of CCS, but playing an opposite role: while synchronisation adds, restriction removes certain transitions. Algebraically, the two operators have some interesting laws in common. In terms of the Petri net semantics of PBC, it will be seen that the distinction between control flow and communication interface operators manifests itself in a very simple way: the latter (relabelling, synchronisation, restriction and scoping) modify transitions, while the former (sequence, choice, parallel composition and iteration) modify places.

2.7 Other operators

Most of the remaining operators of the basic PBC are akin to their counterparts in CCS. The class of control flow connectives, which includes sequential composition and parallel composition, also includes alternative composition (often called 'choice', for short) and iteration. Moreover, PBC also allows the expression of recursion. The class of communication interface connectives, which includes synchronisation, also includes basic relabelling, restriction and scoping.

The choice between two expressions E and F, denoted by $E \,\square\, F$, specifies behaviour which is, in essence, the union of the behaviours of E and F. That is, any behaviour which is a behaviour of E or F is an acceptable behaviour of $E \,\square\, F$, and no other is. Thus the choice operator, as its counterpart in CCS, allows one to choose nondeterministically between two possible sets of behaviours. The only difference with the corresponding CCS operator, $+$, will be syntactic; the PBC notation following the syntactic convention, $\square$, of Dijkstra's guarded commands [18].

The iteration operator of the basic PBC obeys the syntax $[E * F * E']$, where E is the initialisation (which may be executed once, at the beginning), F is the body (whose execution may be repeated arbitrarily many times, after initialisation), and E' is the termination (which may be executed at most once, at termination). Recursion allows recursion variables to be defined via equations. For example, $X \stackrel{\mathrm{df}}{=} a;X$ specifies the behaviour of X to be an indefinite succession of the executions of a. Recursion in PBC is more general than its counterpart in CCS, due to the possibility of unguardedness.

Basic relabelling (simply called 'relabelling' in CCS) is defined with respect to a function f which consistently renames action particles; $E[f]$ has all the behaviours of E, except that their constituents are relabelled according to f. This operation is theoretically relevant only with regard to recursion where it adds the possibility of using an infinite number of action particles,[2] and because, as we will show in due course, it is the simplest version of a very general mechanism which we will call 'relabelling', of which all the communication interface operations mentioned so far are special cases.

Restriction is also, as synchronisation, defined with respect to an action particle a. For example, $E \;\mathsf{rs}\; a$, by definition, has all the behaviours of E except those that involve a or $\widehat{a}$. Using restriction, we may give a simple example of an expression whose terminal state is not reachable from the initial state. Consider $F = (\{a\} \;\mathsf{rs}\; a)$. Then $\underline{F}$ is not reachable from $\overline{F}$ because, by the definition of restriction, the latter can make no move at all. However, both $\overline{\{a\} \;\mathsf{rs}\; a}$ and $\underline{\{a\} \;\mathsf{rs}\; a}$ are well defined dynamic PBC expressions.

Scoping, $[a : E]$, is a derived operator which consists of synchronisation followed by restriction (on the same action particle), i.e., $[a : E] = (E \;\mathsf{sy}\; a) \;\mathsf{rs}\; a$; its importance is in describing blocks in a programming language, as we will sketch in the next section, and more fully in section 6.

2.8 Modelling concurrent programming languages

After motivating and describing the multiaction feature of PBC and giving informal definitions of its basic operators, we now discuss one of their applications. Recall that it was our second motivation (besides the wish to have a close relationship with Petri nets) for PBC to be able to give a flexible semantics to concurrent programming languages. In this section, we discuss this problem informally. Consider, for example,

[2] As for the equation $X \stackrel{\mathrm{df}}{=} \{0\};X[addone]$, where *addone* is a relabelling function adding 1 to each action particle $k \in \mathbf{N}$. This behaviour generates the sequence $\{0\}\{1\}\{2\}\{3\}\ldots$.

the following fragment of a concurrent program:

$\ldots$ **begin var** $x : \{0,1\}$; $[\![x{:=}x \oplus 1]\!] \,||\, [\![x{:=}y]\!]$ **end** $\ldots$

where y is assumed to be declared with type $\{0,1\}$ in some outer block, $\oplus$ denotes the addition modulo 2, $||$ denotes 'shared variable parallelism' (as variable x occurs on both sides of the $||$), and $[\![\ldots]\!]$ delineate atomic actions. Consider constructing an appropriate, and also as small as possible, Petri net describing this block. The construction of this net should also be compositional, that is, it should be composed from nets derived for its three constituents, the declaration **var**$x : \{0,1\}$ and the two atomic assignments, $[\![x{:=}x \oplus 1]\!]$ and $[\![x{:=}y]\!]$, and itself be composable, i.e., usable in further compositions with similar constituents coming, for example, from outer blocks.

Using the basic PBC and its Petri net semantics, this problem can be solved in the following way. We will allow some action particles to be indexed terms of the form x_{vw} and $\hat{x}_{vw}$ where $v,w \in \{0,1\}$. Each such term denotes the change of the value of the program variable x from v to w, or the test of the value of x if $v{=}w$. Using such action particles, the following could be a reasonable translation of the two assignments into PBC expressions:

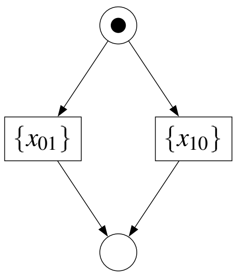
$$\begin{array}{lcl} [\![x{:=}x{\oplus}1]\!] & \leadsto & \{x_{01}\} \,\square\, \{x_{10}\} \\ [\![x{:=}y]\!] & \leadsto & \{x_{00}, y_{00}\} \,\square\, \{x_{10}, y_{00}\} \,\square\, \{x_{01}, y_{11}\} \,\square\, \{x_{11}, y_{11}\}. \end{array} \tag{2}$$

The PBC expression on the right-hand side of the first line expresses that 'either x could be 0, and then it is turned into 1, or x could be 1, and then it is turned into 0', which clearly is the semantics of $[\![x{:=}x{\oplus}1]\!]$ if x is a variable of type $\{0,1\}$. Note that, as opposed to the multiactions in the first line, the two-element multiactions in the second line refer to both x and y, and thus are non-singletons, because $[\![x{:=}y]\!]$ involves both variables. Each multiaction on the second line should be interpreted as denoting two simultaneously executable accesses to the variables, either to check (y) or to change the value (x). For instance, the multiaction $\{x_{10}, y_{00}\}$ denotes the value of y being checked to be 0 and, simultaneously, the value of x being changed from 1 to 0. The expressions for $[\![x{:=}x \oplus 1]\!]$ and $[\![x{:=}y]\!]$ given in (2) have corresponding Petri nets with respectively two and four transitions labelled by the corresponding multiactions, as shown in figure 5.

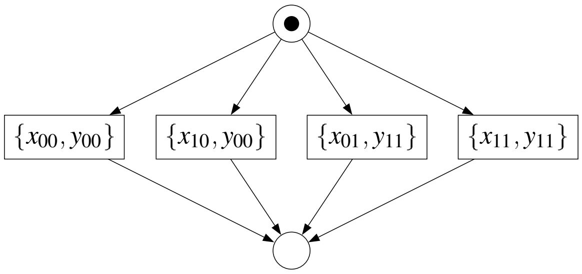
Let us now consider the net corresponding to the declaration **var** $x : \{0,1\}$. It will involve transitions of the form $\hat{x}_{vw}$ for all possible values $v,w \in \{0,1\}$, where $\hat{x}_{vw}$ is the conjugate of x_{vw}. Like x_{vw}, $\hat{x}_{vw}$ also denotes the change of value of x from v to w. We use the conjugates here because this ensures that the net of the variable(s) can be put in parallel with the net of the atomic action(s), and both nets can be synchronised. Such a synchronisation is needed in order to describe a block.

What was just described is the principal use of conjugation in the PBC semantics of programming languages; we will always assume that the command part of a block uses 'unhatted' action symbols, while the declaration part uses their 'hatted' (conjugated) versions in order to describe accesses to variables. Eventually, a block will be described by putting command part and declaration part in parallel and then synchronising and restricting (hence scoping) over all variables that are local to it. This will leave only non-local accesses to be still visible in the communication interface, i.e., in the transitions.

The net describing **var** $x : \{0,1\}$ is shown – in slightly abbreviated form – in figure 6.



Petri net translation of $[\![x{:=}x\oplus 1]\!]$ (x binary)



Petri net translation of $[\![x{:=}y]\!]$ (both x and y binary)

Figure 5: Petri net representation of $[\![x{:=}x\oplus 1]\!]$ and $[\![x{:=}y]\!]$.

Here it is arbitrarily assumed that the current value of x is 1, and thus the net contains a token on the corresponding place.

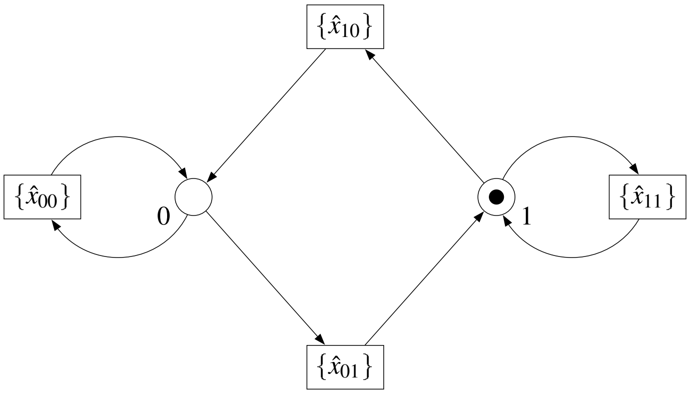


Figure 6: (Part of the) Petri net representing a binary variable x.

In order to describe the block structure of the program (with x being a local and y being a global variable of the block), the net of $\mathbf{var}\, x : \{0,1\}$ needs to be synchronised with the nets of $[\![x{:=}x \oplus 1]\!]$ and $[\![x{:=}y]\!]$ selectively. That is, a transition labelled $\hat{x}_{10}$ coming from the declaration of x is synchronised with a transition labelled $\{x_{10}, y_{00}\}$ coming from the action $[\![x{:=}y]\!]$ using and consuming the conjugate pair $(\hat{x}_{10}, x_{10})$ but retaining y_{00} (in fact, as we have seen, this is built into the definition of synchronisation). The resulting transition has the label $\{y_{00}\}$. After that, all the $\hat{x}_{uv}$ and x_{uv} action particles are wiped out by applying the PBC restriction operator, since variables are not known outside their declaring block. The corresponding net after synchronisation and restriction is shown in figure 7.
The transitions labelled $\{y_{vv}\}$ can then be synchronised with transitions labelled $\{\hat{y}_{vv}\}$ coming from the declaration of y in an outer block, using and consuming conjugate pairs $(y_{vv}, \hat{y}_{vv})$ and yielding transitions labelled by $\emptyset$, after a 3-way synchronisation. During the above translation process, we thus have seen an effective application of the multiway synchronisation mechanism.

3 Syntax and Operational Semantics

3.1 Basic PBC syntax

By definition, a *(basic) static PBC expression* is a word generated by the syntax

$$E ::= \alpha \mid X \mid E \| E \mid E \square E \mid E;E \mid [E{*}E{*}E] \mid E[f] \mid E \;\mathsf{sy}\; a \mid E \;\mathsf{rs}\; a \mid [a{:}E] \tag{3}$$

possibly with parentheses, used – if needed – to resolve ambiguities. In (3), α is an element of $\mathsf{Lab}_{\mathrm{PBC}} = \mathsf{mult}(\mathrm{A}_{\mathrm{PBC}})$ (cf. Section 2.5 where the set $\mathrm{A}_{\mathrm{PBC}}$ and the multiset $\mathsf{Lab}_{\mathrm{PBC}}$ were first used); X is a member of a set $\mathcal{X}$ of predefined *recursion variables*,[3]

[3]Not to be confused with program variables.

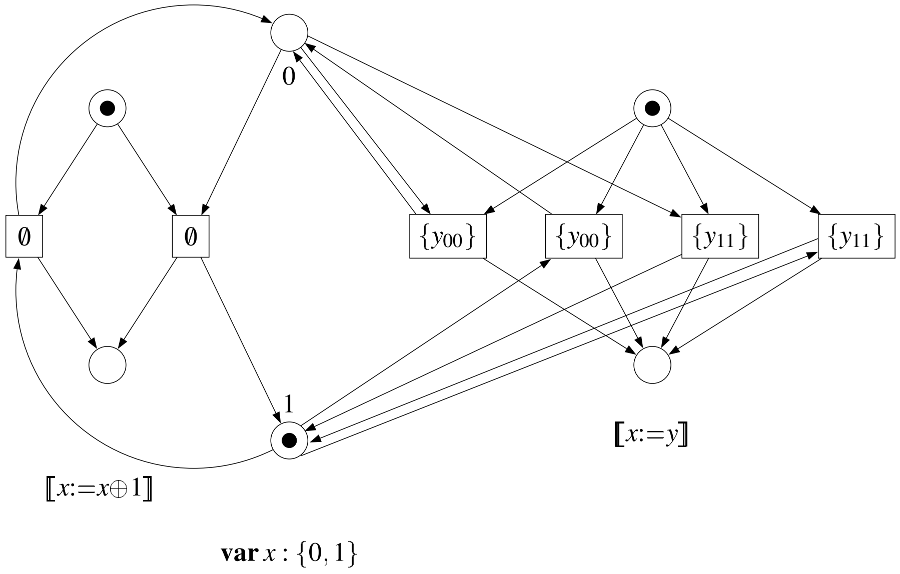


Figure 7: Petri net representation of a small program fragment.

ranged over by $X, Y, Z, \ldots$; a is an element of $\mathsf{A}_{\mathrm{PBC}}$; and f is a *relabelling function* from $\mathsf{A}_{\mathrm{PBC}}$ to $\mathsf{A}_{\mathrm{PBC}}$ that preserves conjugates, that is, by definition, for any action particle $b \in \mathsf{A}_{\mathrm{PBC}}$, $f(\widehat{b}) = \widehat{f(b)}$.

An expression of the form α is called a *basic action* or *multiaction*. The PBC operators fall into two categories. The binary operators ; (sequence), $\Box$ (choice) and $\|$ (disjoint parallelism), and the ternary operator $[\, * * \,]$ (loop with initialisation and termination) will be called the *control (flow) connectives*. The unary operators $[f]$ (basic relabelling), $\mathsf{sy}\ a$ (synchronisation), $\mathsf{rs}\ a$ (restriction) and $[a :]$ (scoping) will be called *(communication) interface operators* or *(generalised) relabelling operators*. The *finite PBC* is obtained by excluding the iteration and recursion operators. The following expressions of the finite PBC will serve as our running examples:

$$\begin{array}{lll} E_0 = \emptyset & E_1 = a;(b \Box c) & E_2 = ((a;b) \Box c)\ \mathsf{rs}\ c \\ E_3 = (a\|b)\ \mathsf{rs}\ b & E_4 = ((a\|\{\widehat{a},\widehat{a}\})\|a)\ \mathsf{sy}\ a. & \end{array} \tag{4}$$

To avoid excessive bracketing, we will often use the convention of replacing a singleton (multi)set such as $\{a\}$ by its only element, in this case a. For instance, $(a;b) \Box c$ simplifies the expression $(\{a\};\{b\}) \Box \{c\}$. This will be done only in example expressions, in the corresponding nets, and when referring to such examples, but never if the simplification might give rise to a confusion.

Let E be a static expression, $a \in \mathsf{A}_{\mathrm{PBC}}$, and f be a relabelling function from $\mathsf{A}_{\mathrm{PBC}}$ to $\mathsf{A}_{\mathrm{PBC}}$, as before. A *dynamic PBC expression* is a word generated by the syntax

$$\begin{array}{rl} G \ ::= & \overline{E} \mid \underline{E} \mid G\|G \mid G \Box E \mid E \Box G \mid G;E \mid E;G \mid [G{*}E{*}E] \mid \\ & [E{*}G{*}E] \mid [E{*}E{*}G] \mid G[f] \mid G\ \mathsf{sy}\ a \mid G\ \mathsf{rs}\ a \mid [a : G] \end{array} \tag{5}$$

possibly with parentheses. Note that an expression such as $G;H$ is not a syntactically valid dynamic expression. The clause $G;E$ means that the first part of the sequence is currently active (which includes its initial state and its final state), while the second part is currently dormant. The other clause, $E;G$, means that the second part of the sequence is currently active, while the first part is dormant. A similar remark holds for the choice composition $\square$ and for iteration $[**]$; we require syntactically that only one of the parts of a choice or iteration expression is active. By contrast, concurrent composition $||$ is allowed – and even required – to satisfy the property that *both* of its parts are active in a dynamic expression.

Let E be any static expression of the basic PBC. Then $\overline{E}$ is the dynamic expression which is associated with E in a canonical way, in the sense that $\overline{E}$ describes the initial state of E. Henceforth, whenever we speak of 'the behaviour of E', we mean the behaviour that is generated by $\overline{E}$, the initial state of E. In the following, we will characterise the behaviour exhibited by the example expressions E_0–E_4 defined above, by which we will mean the behaviour generated by their initial dynamic counterparts, $\overline{E_0}$–$\overline{E_4}$.

E_0 (more precisely, the dynamic expression $\overline{E_0}$) can do a 'silent' $\emptyset$-move and terminate. E_1 can do an a-move, followed either by a b-move or by a c-move (and terminate). E_2 can make an a-move followed by a b-move and terminate, but cannot make a c-move. E_3 can make an a-move but it cannot make a b-move, nor can it terminate. The last expression, E_4, can make the same moves as the expression $(a||\{\widehat{a},\widehat{a}\})||a$ and, in addition, three synchronisations: an $\widehat{a}$-move synchronising the left and the middle components of the parallel composition; an $\widehat{a}$-move synchronising the middle and the right-hand side components; and a silent move synchronising all three components (and terminate).

To express recursion, PBC — as other process algebras — uses syntactic (or hierarchical) variables, forming the set $\mathcal{X}$. Such variables can appear on the left-hand sides as well as on the right-hand sides of recursive equations at any position where a multiaction α is also allowed. For instance, $X \stackrel{\mathrm{df}}{=} \{a\};X$ introduces a variable X together with a defining equation in which X occurs on the right-hand side, i.e., it is a true recursion. Recursion may result from more complex structures, such as in the expression $X \stackrel{\mathrm{df}}{=} Y||Z$ where Y and Z are defined thus:

$$Y \stackrel{\mathrm{df}}{=} \{a,b\};Z \text{ and } Z \stackrel{\mathrm{df}}{=} \{a,a\} \square (Y;Z).$$

The latter is a *(recursive) system of equations* as we have three equations, one for each of X, Y and Z. There will be exactly one defining equation per recursion variable. A system of equations can be turned into an expression of the basic PBC by choosing any of the equations – typically the first one – and decreeing that the behaviour of the corresponding variable is the behaviour of the expression; in the above example, X would determine the behaviour of the expression.

We deal with recursion in its most general (possibly non-guarded) form, i.e., we do not introduce any syntactic restrictions on the position of variables in an expression. They may occur at the beginning, at the end, or in the middle of an expression, or even everywhere at once (such as, for instance, X in the defining equation $X \stackrel{\mathrm{df}}{=} X \square X$). Equations may always be rewritten (using fresh variables) to avoid complex right-hand sides. Consider, for instance, the defining equation $Y \stackrel{\mathrm{df}}{=} \{a\};(\{b\} \square Y)$. This equation may be rewritten as the system: $Y \stackrel{\mathrm{df}}{=} \{a\};Y'$, $Y' \stackrel{\mathrm{df}}{=} \{b\} \square Y$.

3.2 Structured operational semantics

Structured operational semantics (SOS, [39]) is a well established approach to defining the set of possible moves of a dynamic expression – or, more technically speaking, an operational semantics – of a process algebra. SOS consists of a set of axioms and derivation rules from which evolutionary behaviours of (dynamic) expressions can be derived.

Let *exp* and *exp'* be two dynamic expressions of a process algebra (recall that in CCS, all expressions are dynamic objects, while in PBC, there exists a dedicated syntax for them). The possible evolutions leading from *exp* to *exp'* are given in the form of triples

$$evol \quad = \quad exp \xrightarrow{\quad action \quad} exp' . \tag{6}$$

The intended meaning is that, when started in a state described by *exp*, *action* may be performed, and after that, a state described by *exp'* is reached. The SOS axioms specify a set of basic evolutions (of the kind shown above), while the derivation rules take the form

$$\frac{evol_1\ ,\ \ldots\ ,\ evol_n}{evol}\,cond$$

meaning that if the evolutions $evol_1,\ldots,evol_n$ (the *premises*) are already derived and the condition *cond* (on the parameters of the evolutions $evol_i$) is satisfied, then the evolution *evol* (the *conclusion*) may be derived; if *cond* is missing, it means no extra condition is required. Typically, the $evol_i$'s give evolutions of subterms of a larger expression whose evolution is described by *evol*, i.e., the rule has the form

$$\frac{exp_1 \xrightarrow{\quad action_1 \quad} exp'_1\ ,\ \ldots\ ,\ exp_n \xrightarrow{\quad action_n \quad} exp'_n}{\mathsf{op}(exp_1,\ldots,exp_n) \xrightarrow{\quad action \quad} \mathsf{op}(exp'_1,\ldots,exp'_n)}\,cond \tag{7}$$

where op is one of the operators of the process algebra, and *action* is typically determined by some transformation applied to the individual evolutions $action_i$. Axioms may be viewed as derivation rules without premises. The compositionality of the semantics arises from the fact that the derivation rules relate the behaviour of an expression to the behaviours of its components. For instance, CCS has an axiom $a.E \xrightarrow{a} E$ which means that by performing action a, expression $a.E$ is transformed into expression E, and a derivation rule

$$\frac{E \xrightarrow{a} E'\ ,\ F \xrightarrow{\widehat{a}} F'}{E \,|\, F \xrightarrow{\tau} E' \,|\, F'} \tag{8}$$

(often called the 'τ-rule') which means that if two conjugate actions are concurrently executable, then they are executable together as a silent action.

Let us now have a closer look at, and adapt, the rule schema for the purposes of PBC. We must consider multiactions, such as $\emptyset$, $\{a\}$, $\{\widehat{a}\}$, $\{\widehat{a},b\}$, or $\{a,a,b\}$. In addition, the close relationship with Petri nets, and the separation of concurrent composition and synchronisation allows us to tune the SOS semantics in such a way that it describes not just behaviour, but *concurrent behaviour*. That is, what has been called an 'action' above, will now be formalised as a *step* of multiactions, i.e., a finite multiset of finite multisets of action particles, Γ, as follows:

$$\Gamma \in \mathsf{MLab} = \mathsf{mult}(\mathsf{Lab}_{\mathsf{PBC}}) = \mathsf{mult}(\mathsf{mult}(\mathsf{A}_{\mathsf{PBC}})).$$

We shall use $\Gamma, \Delta, \ldots$ to range over steps. For the dynamic PBC expressions, we thus aim at axiomatising a relation $G \xrightarrow{\Gamma} H$ meaning that G can execute a step Γ (i.e., execute simultaneously, or concurrently, all the multiactions forming the step) and yield H. Note that the *empty step* $\Gamma = \emptyset$ is allowed; it denotes 'no action', and thus, we would expect to have $G \xrightarrow{\emptyset} G$ in general. The empty step should not be confused with the singleton step $\Gamma = \{\emptyset\}$ consisting only of the silent action $\emptyset$; in general, we do not have $G \xrightarrow{\{\emptyset\}} G$. More generally, we shall be interested in the *step sequence* semantics, i.e., in derivations of the form

$$G \xrightarrow{\Gamma_1\Gamma_2\ldots\Gamma_n} H$$

where $\Gamma_1\Gamma_2\ldots\Gamma_n$ is a sequence of steps, implying that there are derivations

$$G \xrightarrow{\Gamma_1} G_1 \xrightarrow{\Gamma_2} G_2 \quad \cdots \quad G_{n-1} \xrightarrow{\Gamma_n} H.$$

In this respect, the empty step $\Gamma = \emptyset$ will act as the neutral element in step sequences so that, for instance, the two derivations

$$G \xrightarrow{\emptyset\Gamma_1\emptyset\emptyset\Gamma_2\emptyset} H \quad \text{and} \quad G \xrightarrow{\Gamma_1\Gamma_2} H$$

will be equivalent. Moreover, we shall consider the following relation between two dynamic expressions:

$$G \equiv H \quad \Leftrightarrow \quad G \xrightarrow{\emptyset} H. \tag{9}$$

Any dynamic expressions related by $\equiv$ will be called *structurally equivalent*. The following set of general *inaction rules* captures the main properties of empty steps, and hence also of the $\equiv$-relation:

IN1	$G \xrightarrow{\emptyset} G$	IN2	$\dfrac{G \xrightarrow{\emptyset} H}{H \xrightarrow{\emptyset} G}$
ILN	$\dfrac{G \xrightarrow{\emptyset\Gamma} H}{G \xrightarrow{\Gamma} H}$	IRN	$\dfrac{G \xrightarrow{\Gamma\emptyset} H}{G \xrightarrow{\Gamma} H}$

The rule IN1 (INaction rule 1) implies the reflexivity of the relation $\equiv$, and IN2 implies its symmetry. The next two rules, ILN (Inaction is Left Neutral) and IRN (Inaction is Right Neutral), imply the transitivity of the relation (since in particular, $G \xrightarrow{\emptyset\emptyset} H$ implies $G \xrightarrow{\emptyset} H$). Thus, $\equiv$ is indeed an equivalence relation.

Beside the various semantical operators of PBC, we shall also use a syntactical one: if G is a dynamic expression, $\lfloor G \rfloor$ will denote its underlying static expression, obtained from G by dropping all its overbars and underbars. For instance, $\lfloor(\underline{a};b)\rfloor = (a;b)$.

3.2.1 Equivalence notions

The question of whether formal descriptions of two concurrent system can be regarded as behaviourally equivalent is recurring in one guise or another throughout the literature. For instance, in order to be able to say that a binary operator is commutative or associative, one needs to state that various expressions constructed from it are behaviourally equivalent. However, various equivalence notions may be considered for the purpose. As this discussion is independent of the actual definition of *action* in (6), we may conduct it in advance.

A first candidate for such an equivalence notion could be the $\equiv$-relation, as it is an equivalence, and a behavioural one in a strong sense since, due to ILN and IRN, if

$$G' \equiv G \xrightarrow{\Gamma_1\Gamma_2\ldots\Gamma_n} H \equiv H'$$

then the following are satisfied:

$$G' \xrightarrow{\Gamma_1\Gamma_2\ldots\Gamma_n} H \qquad G \xrightarrow{\Gamma_1\Gamma_2\ldots\Gamma_n} H' \qquad G' \xrightarrow{\Gamma_1\Gamma_2\ldots\Gamma_n} H'.$$

However, this relation is too strong for our purposes since it is not necessarily the case that two dynamic expressions that correspond to the same Petri net are $\equiv$-equivalent. In particular, $\overline{a;(b;c)} \not\equiv \overline{(a;b);c}$, since the underlying expressions are not exactly the same, while (as it will occur) $G \equiv H \Rightarrow \lfloor G \rfloor = \lfloor H \rfloor$; however, their Petri net translations will be isomorphic, and their equivalence is what is required for the sequential composition to be associative.

On the other hand, it would not be enough to only require that the step sequences be the same for the two expressions under consideration. Indeed, if we denote by stop a hierarchical variable with the defining equation $\mathsf{stop} \overset{\mathrm{df}}{=} \mathsf{stop}$, so that $\overline{\mathsf{stop}}$ is only ever able to perform the looping empty move $\overline{\mathsf{stop}} \xrightarrow{\emptyset} \overline{\mathsf{stop}}$, then $\overline{a;a}$ and $\overline{(a;a) \Box (a;\mathsf{stop})}$ will lead to the same nonempty move sequences $\{\{a\}\}$ and $\{\{a\}\}\{\{a\}\}$; but it would be inappropriate to consider that the two expressions are equivalent since the latter may be blocked after the first execution of a (if the second branch of the choice is followed), while this may not happen for the former. Similarly, $a;(b \Box c)$ is not equivalent to $(a;b) \Box (a;c)$. Hence, it is necessary to take into account the *branching structure* of the sequences of moves.

A common way to represent branching structures is to use *(labelled) transition systems*. A transition system is usually defined as a quadruple $\mathsf{ts} = (V, L, A, v_{in})$ which consists of a set V of states; a set L of arc labels; a set $A \subseteq V \times L \times V$ of arcs;[4] and an initial state v_{in}. Being essentially 'model independent', transition systems are often the preferred tool to compare various semantics. For if we associate a transition system both to an SOS-semantics of a (static or dynamic) PBC expression and to an associated (unmarked or marked) Petri net, it will be possible to state that the two semantics are equivalent, through equivalences defined at the transition system level.

A first attempt to associate a labelled transition system with a dynamic expression G could be to consider the tuple $([G\rangle, \mathsf{MLab}, A, G)$ where the set of states $[G\rangle$ is the smallest set of expressions containing G such that if $H \in [G\rangle$ and $H \xrightarrow{\Gamma} J$ then $J \in [G\rangle$,

[4] In the literature these are called transitions (hence the name: transition system); however, we shall reserve this term to mean a component of a Petri net.

and the arcs are given by:

$$A \;=\; \{(H,\Gamma,J) \mid H \in\; [G\rangle \;\wedge\; \Gamma \in \mathsf{MLab} \;\wedge\; H \xrightarrow{\Gamma} J\}.$$

Another possibility would be to replace, in the previous transition system, the set $[G\rangle$ by the set of its $\equiv$-equivalence classes (and the initial state G by its $\equiv$-class), since two $\equiv$-equivalent expressions have the same behaviours and correspond to two views of the same system. However neither of these two definitions would faithfully capture the full complexity of the intended behaviours. Indeed, one could reach the conclusion that the two expressions $\overline{\mathsf{stop}}$ and $\underline{\mathsf{stop}}$ are equivalent, since they only allow empty moves. But it would not be advisable to do so since $\overline{\mathsf{stop}};a$ and $\underline{\mathsf{stop}};a$ are certainly not equivalent; the latter allows one to perform an a-move while the former only allows empty moves. What we really need is not only a behavioural equivalence, but more exactly a behavioural congruence, i.e., an equivalence, say $\sim$, which is preserved by every operator op of the algebra:

$$G_1 \sim H_1 \;,\ldots,\; G_n \sim H_n \quad\Longrightarrow\quad \mathsf{op}(G_1,\ldots,G_n) \sim \mathsf{op}(H_1,\ldots,H_n).$$

The example above shows that, in order to achieve this, it is necessary to distinguish the terminal expressions (and the $\equiv$-equivalent ones) from the non-terminal ones. But this is not enough, since $\underline{a}$ and $\underline{b}$ are both terminal expressions allowing empty moves only, but the looping constructs $[c * \underline{a} * c]$ and $[c * \underline{b} * c]$ are certainly not equivalent: the latter allows one to perform a series of b-moves (terminated by a c-move), while the former only allows a series of a-moves (terminated by a c-move). Hence, it is also necessary to take into account the fact that some constructs of the PBC algebra connect terminal (sub-)expressions to the corresponding initial ones. And, by a symmetric argument, it is desirable to distinguish the initial expressions and to take into account the fact that the looping constructs also connect initial (sub-)expressions to the corresponding terminal ones.

All these features may be captured by the following device: when constructing the labelled transition system associated with an expression (but only for that purpose), we shall artificially augment the action set by two special elements, redo and skip, and add two rules

$$\overline{E} \xrightarrow{\{\mathsf{skip}\}} \underline{E} \quad\text{ and }\quad \underline{E} \xrightarrow{\{\mathsf{redo}\}} \overline{E}.$$

That is, the initial and terminal expressions will be behaviourally distinguished from other ones in that they will allow $\mathsf{skip}/\mathsf{redo}$ moves. Then, with each dynamic expression G we shall associate the labelled transition system $\mathsf{lbts}_G = (\,[G\rangle_{\mathsf{sr}}\,, \mathsf{MLab}_{\mathsf{sr}}, A, G)$ where $\mathsf{MLab}_{\mathsf{sr}} = \mathsf{mult}(\mathsf{Lab}) \cup \{\{\mathsf{skip}\},\{\mathsf{redo}\}\}$ is the set of augmented move labels,

$$[G\rangle_{\mathsf{sr}} \;=\; \{H \mid \exists \Gamma_1,\Gamma_2,\ldots,\Gamma_n \in \mathsf{MLab}_{\mathsf{sr}} \;:\quad G \;\xrightarrow{\;\Gamma_1\Gamma_2\ldots\Gamma_n\;}\; H\,\}$$

is the set of dynamic expressions reachable from G using the augmented rules, and the arcs are given by:

$$A \;=\; \{(H,\Gamma,J) \mid H \in\; [G\rangle_{\mathsf{sr}} \;\wedge\; \Gamma \in \mathsf{MLab}_{\mathsf{sr}} \;\wedge\; H \xrightarrow{\Gamma} J\}.$$

We shall also associate with G a *reduced* labelled transition system

$$\mathsf{lbts}_G^{rdc} \;=\; (\{[H]_\equiv \mid H \in\; [G\rangle_{\mathsf{sr}}\,\}, \mathsf{MLab}_{\mathsf{sr}}, \mathcal{A}, [G]_\equiv)$$

where $[H]_\equiv$ denotes the equivalence class of a dynamic expression H with respect to the $\equiv$-equivalence, and the set of arcs is given by:

$$\mathcal{A} = \{([H]_{\equiv}, \Gamma, [J]_{\equiv}) \mid H \in [G\rangle_{\mathsf{sr}} \wedge \Gamma \in \mathsf{MLab}_{\mathsf{sr}} \wedge H \xrightarrow{\Gamma} J\}.$$

Moreover, with each static expression E, we shall associate the following two labelled transition system: $\mathsf{lbts}_E = \mathsf{lbts}_{\overline{E}}$ and $\mathsf{lbts}_E^{rdc} = \mathsf{lbts}_{\overline{E}}^{rdc}$.

The strongest notion of behavioural equivalence usually defined for labelled transition systems is isomorphism. Two labelled transition systems, (V, L, A, v_{in}) and (V', L', A', v'_{in}) are *isomorphic* if there is a bijection $\mathsf{iso} : V \to V'$ such that $\mathsf{iso}(v_{in}) = v'_{in}$ and

$$A' = \{(\mathsf{iso}(v), l, \mathsf{iso}(w)) \mid (v, l, w) \in A\}.$$

But such an isomorphism is uselessly restrictive since, for instance, the equation $X \stackrel{\mathrm{df}}{=} a\|b$ would 'artificially' increase the $\equiv$-class of $\overline{a\|b}$ and hence add an extra node to the transition system $\mathsf{lbts}_{\overline{a\|b}}$, but it would not affect the expression $\overline{b\|a}$; yet we would expect these two expressions to be equivalent. This leads to the definition that two expressions G and H (being both either static or dynamic) are *lbts-isomorphic*, $G \cong H$, if lbts_G^{rdc} and lbts_H^{rdc} are isomorphic transition systems.

But isomorphism is not the only way to obtain interesting congruences. We shall also examine what in CCS terminology is called strong equivalence.[5] Two labelled transition systems, (V, L, A, v_{in}) and (V', L', A', v'_{in}), are *strongly equivalent* if there exists a relation $Q \subseteq V \times V'$, itself called a *strong bisimulation*, such that $(v_{in}, v'_{in}) \in Q$, and if $(v, v') \in Q$ then

- If $(v, l, w) \in A$ then, for some $w' \in V'$, $(v', l, w') \in A'$ and $(w, w') \in Q$.
- If $(v', l, w') \in A'$ then, for some $w \in V$, $(v, l, w) \in A$ and $(w, w') \in Q$.

Two expressions G and H (being both either static or dynamic) will be *strongly equivalent*, $G \approx H$, if so are lbts_G and lbts_H. Here, we used lbts instead of lbts^{rdc} since lbts_G and lbts_H are strongly equivalent if and only if lbts_G^{rdc} and lbts_H^{rdc} are strongly equivalent. It may also be observed that $\equiv \subset \cong \subset \approx$, i.e., for all expressions G and H,

$$G \equiv H \quad \Rightarrow \quad G \cong H \quad \Rightarrow \quad G \approx H,$$

and that the inclusions are strict. The reader will also be able to check that $\equiv$, $\cong$ and $\approx$ are indeed congruences for the various PBC operators, when we shall define the evolution rules for them in the later part of this section. This is in contrast to step equivalence (sometimes called 'step trace equivalence'): call G and H *step equivalent* if the step sequences generated by them are the same. For instance,

$$G = \overline{(a \,\square\, b)} \quad \text{and} \quad H = \overline{[c : c\|(\{\widehat{c}, a\} \,\square\, b)]}$$

are step equivalent,[6] but $G;d$ and $H;d$ are not, since $G;d$ can generate a step sequence $\{b\}\{d\}$ which cannot be generated by $H;d$. Thus, in a sequential context, G and H may not be freely exchanged for one another.

We now will specify the structured operational semantics, according to the general scheme just described, for the various kinds of (dynamic) PBC expressions. We will go through the basic PBC constants and operators one by one in the order given in the syntax, except that we treat the two infinite operators (iteration and recursion) last. Each

[5] Weak equivalences and congruences, obtained by treating the silent $\{\emptyset\}$-moves as a new kind of empty moves, may also be defined, but we shall not treat them here.

[6] The reader may need to wait until section 3.2.5 in order to fully appreciate this example.

time, we will give inaction rules and derivation rules, as appropriate. We finally stress that no derivation considered below involves the two special actions, skip and redo.

3.2.2 Elementary actions

The operational semantics of $\alpha \in \mathsf{A}_{\mathrm{PBC}}$ is given by the following *action rule*:

AR	$\overline{\alpha} \xrightarrow{\{\alpha\}} \underline{\alpha}$

This rule specifies that α corresponds to an atomic transition (i.e., a transition in the Petri net sense). For instance, we may derive the following evolutions for the first example expression $E_0 = \emptyset$, starting from its initial dynamic version:

$$\overline{\emptyset} \xrightarrow{\emptyset} \overline{\emptyset}\ \text{(IN1)} \quad \text{and} \quad \overline{\emptyset} \xrightarrow{\{\emptyset\}} \underline{\emptyset}\ \text{(AR)}.$$

Notice the difference between the first derivation (inaction) and the second derivation (silent action). The former maintains the initial state of E_0 while the latter transforms the initial state of E_0 into its terminal state and denotes an actual execution of a silent transition.

3.2.3 Parallel composition, choice, and sequential composition

The operational semantics of parallel composition is driven by three rules :

IPAR1	$\overline{E\|\|F} \xrightarrow{\emptyset} \overline{E}\|\|\overline{F}$
PAR	$\dfrac{G \xrightarrow{\Gamma} G'\ ,\ H \xrightarrow{\Delta} H'}{G\|\|H \xrightarrow{\Gamma+\Delta} G'\|\|H'}$
IPAR2	$\underline{E}\|\|\underline{F} \xrightarrow{\emptyset} \underline{E\|\|F}$

These rules combine two inaction rules (IPAR1 and IPAR2) with one *context*, or *derivation*, rule (PAR) in which $\Gamma+\Delta$ denotes the multiset sum of Γ and Δ. The PAR rule should be interpreted as follows. If G can make a Γ step to become G', and H can make a Δ step to become H', then we can infer that $G||H$ can make a $\Gamma+\Delta$ step (thus performing concurrently all the components of both Γ and Δ) to become $G'||H'$. We call it a 'context rule', because its general shape is such that, from the moves of subexpressions of the expression under consideration, a move of the whole expression may be deduced. These moves may be actual 'actions' involving the rule of the previous section, but they also may be inactions. Thus it would be slightly misleading to call PAR an 'action rule'. The inaction rule $\overline{E||F} \xrightarrow{\emptyset} \overline{E}||\overline{F}$ should not be interpreted as denoting anything actually happening in the step sequence sense. Rather, it describes two different views of the same system; the difference between $\overline{E||F}$ and $\overline{E}||\overline{F}$ is that the first specifies an initial state of $E||F$ while the second specifies the parallel composition of two separate initial states, respectively, of E and F, and the inaction rule says that the two views are equivalent. By the inaction rule IN2, it follows that we also have the symmetric rule $\overline{E}||\overline{F} \xrightarrow{\emptyset} \overline{E||F}$. The reason we have given rule IPAR1, instead of its symmetric counterpart, is that in the derivation of behaviour, we will usually use IPAR1, rather than its

reverse. A similar remark holds for IPAR2. It can be shown that $G\|H \cong H\|G$, as well as $G\|(H\|J) \cong (G\|H)\|J$, where G, H and J are such that all the expressions are valid PBC expressions, static or dynamic (the same assumption will be made in the formulation of other properties of PBC operators). We interpret these as properties signifying that $\|$ is *commutative* and *associative*.
The other control operators of the basic PBC (choice composition, sequential composition and iteration) follow the pattern used in the case of parallel composition; their operational semantics consists of a number of inaction rules combined with one or more derivation rules. The rules for the choice composition are:

IC1L	$\overline{E \square F} \xrightarrow{\emptyset} \overline{E} \square F$	IC1R	$\overline{E \square F} \xrightarrow{\emptyset} E \square \overline{F}$
CL	$\dfrac{G \xrightarrow{\Gamma} G'}{G \square F \xrightarrow{\Gamma} G' \square F}$	CR	$\dfrac{H \xrightarrow{\Delta} H'}{E \square H \xrightarrow{\Delta} E \square H'}$
IC2L	$\underline{E} \square F \xrightarrow{\emptyset} \underline{E \square F}$	IC2R	$E \square \underline{F} \xrightarrow{\emptyset} \underline{E \square F}$.

The next set of rules describes the operational semantics of sequential composition:

IS1	$\overline{E;F} \xrightarrow{\emptyset} \overline{E};F$	SL	$\dfrac{G \xrightarrow{\Gamma} G'}{G;F \xrightarrow{\Gamma} G';F}$
IS2	$\underline{E};F \xrightarrow{\emptyset} E;\overline{F}$		
IS3	$E;\underline{F} \xrightarrow{\emptyset} \underline{E;F}$	SR	$\dfrac{H \xrightarrow{\Delta} H'}{E;H \xrightarrow{\Delta} E;H'}$

Choice is commutative, $G \square H \cong H \square G$, and associative, $G \square (H \square J) \cong (G \square H) \square J$. It is also *idempotent*: $E \square E \approx E$, $G \square \lfloor G \rfloor \approx G$ and $\alpha \square \alpha \cong \alpha$. Sequential composition is associative, $G;(H;J) \cong (G;H);J$.

3.2.4 Multiway synchronisation

We will describe three possible ways of defining the synchronisation operator semantically, depending on the amount of new behaviour added. First, consider table 1. SY1 states that $G\ \mathsf{sy}\ a$ can mimic all the steps of G. The second derivation rule, SY2, states that if $G\ \mathsf{sy}\ a$ can make a move to $G'\ \mathsf{sy}\ a$ involving two multiactions, one containing a and the other containing $\widehat{a}$, then $G\ \mathsf{sy}\ a$ can also make a move in which these two actions are combined into a single one, less the synchronising pair, also leading to $G'\ \mathsf{sy}\ a$. If conjugate pairs are suitably distributed, this rule can be applied repeatedly in a single derivation. These rules generalise the CCS rules for parallel composition in the following sense: SY1 is the same rule as its CCS counterpart, and SY2 generalises the 'τ-rule' (8). Synchronisation is commutative, $G\ \mathsf{sy}\ a\ \mathsf{sy}\ b \cong G\ \mathsf{sy}\ b\ \mathsf{sy}\ a$, idempotent, $G\ \mathsf{sy}\ a\ \mathsf{sy}\ a \cong G\ \mathsf{sy}\ a$, and *insensitive to conjugation*, $G\ \mathsf{sy}\ \widehat{a} \cong G\ \mathsf{sy}\ a$.
The rules given previously prohibit what might be called *silent auto-synchronisation*. To see this, consider the PBC expression $\{a,\widehat{a}\}\ \mathsf{sy}\ a$. The step $\{\emptyset\}$ is not allowed for the corresponding initial dynamic expression $\overline{\{a,\widehat{a}\}\ \mathsf{sy}\ a}$, because SY1-SY2 only describe handshake communication between distinct sub-expressions. The following

ISY1	$\overline{E \,\mathsf{sy}\, a} \xrightarrow{\emptyset} \overline{E} \,\mathsf{sy}\, a$
SY1	$\dfrac{G \xrightarrow{\Gamma} G'}{G \,\mathsf{sy}\, a \xrightarrow{\Gamma} G' \,\mathsf{sy}\, a}$
SY2	$\dfrac{G \,\mathsf{sy}\, a \xrightarrow{\{\alpha + \{a\}\} + \{\beta + \{\widehat{a}\}\} + \Gamma} G' \,\mathsf{sy}\, a}{G \,\mathsf{sy}\, a \xrightarrow{\{\alpha + \beta\} + \Gamma} G' \,\mathsf{sy}\, a}$
ISY2	$\underline{E} \,\mathsf{sy}\, a \xrightarrow{\emptyset} \underline{E \,\mathsf{sy}\, a}$

Table 1: The scheme discussed informally in section 2.5.

is a modification making silent auto-synchronisation possible without destroying the scheme for incremental multiway synchronisation:

$$\boxed{\text{ISY1, SY1, SY2, ISY2}} \quad \text{and} \quad \boxed{\text{SY3} \quad \dfrac{G \,\mathsf{sy}\, a \xrightarrow{\{\alpha + \{a, \widehat{a}\}\} + \Gamma} G' \,\mathsf{sy}\, a}{G \,\mathsf{sy}\, a \xrightarrow{\{\alpha\} + \Gamma} G' \,\mathsf{sy}\, a}}$$

The new rule SY3 allows additional steps to be made by a synchronised expression. In particular, the derivation

$$\overline{\{a, \widehat{a}\} \,\mathsf{sy}\, a} \xrightarrow{\{\emptyset\}} \underline{\{a, \widehat{a}\} \,\mathsf{sy}\, a}$$

is now possible. Moreover, since we can combine the rules SY1, SY2 and SY3, we may effectively realise what could be called *multilink-synchronisations*, i.e., synchronisations combining more than one conjugate link between two partners; e.g.,

$$\overline{(\{a, a\} || \{\widehat{a}, \widehat{a}\}) \,\mathsf{sy}\, a} \xrightarrow{\{\emptyset\}} \underline{(\{a, a\} || \{\widehat{a}, \widehat{a}\})} \,\mathsf{sy}\, a$$

holds by SY1, SY2 and SY3 (but not by SY1 and SY2 alone). It can be shown that the commutativity, idempotence and insensitivity to conjugation of the synchronisation operator is preserved by the addition of the rule SY3. We finally consider the following rules:

$$\boxed{\text{ISY1, SY1, SY3, ISY2}} \quad \text{and} \quad \boxed{\text{SY4} \quad \dfrac{G \xrightarrow{\{\alpha\} + \{\beta\} + \Gamma} G'}{G \xrightarrow{\{\alpha + \beta\} + \Gamma} G'}}$$

The SY4 rule can be applied to arbitrary (dynamic) expressions, not just synchronised ones; it means that two (or more, through its repeated use) concurrent actions may always be combined in a single one, offering simultaneously what they both offered separately. For instance, with SY4 we may infer

$$\overline{a || b} \xrightarrow{\{\{a, b\}\}} \underline{a || b}.$$

If we add the rules ISY1, SY1, SY3 and ISY2 to SY4, we obtain the usual synchronisations together with auto-synchronisations and multilink-synchronisations, without the need for SY2. The last synchronisation rule is the strongest possible synchronisation, since it allows arbitrary concurrent actions to be combined into a single one; it has been presented here as the last, 'maximal', in the series of generalisations of the standard CCS rule. Again, the commutativity, idempotence and insensitivity to conjugation of synchronisation is preserved by the addition of rule SY4.

3.2.5 Basic relabelling, restriction and scoping

Let f be a function $f\colon \mathsf{A}_{\mathrm{PBC}} \to \mathsf{A}_{\mathrm{PBC}}$ satisfying $f(\widehat{a}) = \widehat{f(a)}$ for all $a \in \mathsf{A}_{\mathrm{PBC}}$, i.e., f is conjugate-preserving. We lift f to a function from $\mathsf{Lab}_{\mathrm{PBC}}$ to $\mathsf{Lab}_{\mathrm{PBC}}$ (also denoted by f) by the formula $f(\alpha) = \sum_{a\in\mathsf{A}_{\mathrm{PBC}}} \alpha(a)\cdot\{f(a)\}$ where the sum ($\sum$) and the multiplication by a natural number ($\cdot$) have their usual meanings in the multiset domain. After that we lift f to $\mathsf{mult}(\mathsf{Lab}_{\mathrm{PBC}})$, in a similar way: $f(\Gamma) = \sum_{\alpha\in\mathsf{Lab}_{\mathrm{PBC}}} \Gamma(\alpha)\cdot\{f(\alpha)\}$. For the basic relabelling, we then define the following operational semantics:

IR1	$\overline{E[f]} \xrightarrow{\emptyset} \overline{E}[f]$	IR2	$\underline{E}[f] \xrightarrow{\emptyset} \underline{E[f]}$
RR	$\dfrac{G \xrightarrow{\Gamma} G'}{G[f] \xrightarrow{f(\Gamma)} G'[f]}$		

One can show that $G[f][g] \cong G[g\circ f]$ and $G[id] \cong G$, where id is the identity relabelling. As in CCS, the restriction of a PBC expression E with respect to an action particle a is defined as the expression, $E \;\mathsf{rs}\; a$, such that all the actions of E are allowed, except those containing a or $\widehat{a}$:

IRS1	$\overline{E \;\mathsf{rs}\; a} \xrightarrow{\emptyset} \overline{E} \;\mathsf{rs}\; a$	IRS2	$\underline{E} \;\mathsf{rs}\; a \xrightarrow{\emptyset} \underline{E \;\mathsf{rs}\; a}$
RS	$\dfrac{G \xrightarrow{\Gamma} G'}{G \;\mathsf{rs}\; a \xrightarrow{\Gamma} G' \;\mathsf{rs}\; a}$	$\Gamma \in \mathsf{mult}(\mathsf{mult}(\mathsf{A}_{\mathrm{PBC}}\backslash\{a,\widehat{a}\}))$	

Restriction is commutative, $G \;\mathsf{rs}\; a \;\mathsf{rs}\; b \cong G \;\mathsf{rs}\; b \;\mathsf{rs}\; a$, idempotent, $G \;\mathsf{rs}\; a \;\mathsf{rs}\; a \cong G \;\mathsf{rs}\; a$, and insensitive to conjugation, $G \;\mathsf{rs}\; \widehat{a} \cong G \;\mathsf{rs}\; a$.
The scoping of a PBC expression E with respect to a primitive action a, denoted by $[a:E]$, is a derived operator defined as $[a:E] = (E \;\mathsf{sy}\; a) \;\mathsf{rs}\; a$. Its semantics thus follows from that of restriction and synchronisation. Scoping satisfies the same properties as synchronisation and restriction, i.e., it is commutative, idempotent and insensitive to conjugation.

3.2.6 Iteration

The iteration of a PBC expression F with an initialisation E and a termination E' is a new expression denoted syntactically by $[E * F * E']$. Intuitively, it specifies the execution of E, followed (if E does not deadlock before reaching its end and does not enter an infinite loop) by an arbitrary number of executions of F (including zero and infinitely many times), followed possibly (if none of the executions of F deadlocks or loops indefinitely, and F is not executed indefinitely) by the execution of E'. The reason for providing the loop with nonempty initialisation and termination actions will be discussed later. It also corresponds to common programming practice. For example, consider the loop

$$[\![i{:=}0]\!];\ \mathbf{do}\ [\![A[i]{=}0]\!] \rightarrow [\![i{:=}i{+}1]\!]\ \mathbf{od}$$

which specifies a linearly ascending search for a nonzero element of the array A. The loop has an explicit initialisation $[\![i{:=}0]\!]$ and an implicit termination, namely the test for nonzero, $[\![\neg(A[i]{=}0)]\!]$. Hence it corresponds to the following expression:

$$[\ [\![i := 0]\!] * ([\![A[i] = 0]\!] ; [\![i := i+1]\!]) * [\![A[i] \neq 0]\!]\ ].$$

Formally, the operational semantics of the iteration with initialisation and termination is defined as follows:

IIT1	$\overline{[E * F * E']} \xrightarrow{\emptyset} [\overline{E} * F * E']$	IT1	$\dfrac{G \xrightarrow{\Gamma} G'}{[G * F * E'] \xrightarrow{\Gamma} [G' * F * E']}$
IIT2a	$[\underline{E} * F * E'] \xrightarrow{\emptyset} [E * \overline{F} * E']$		
IIT2b	$[E * \underline{F} * E'] \xrightarrow{\emptyset} [E * \overline{F} * E']$	IT2	$\dfrac{G \xrightarrow{\Gamma} G'}{[E * G * E'] \xrightarrow{\Gamma} [E * G' * E']}$
IIT2c	$[E * \underline{F} * E'] \xrightarrow{\emptyset} [E * F * \overline{E'}]$	IT3	$\dfrac{G \xrightarrow{\Gamma} G'}{[E * F * G] \xrightarrow{\Gamma} [E * F * G']}$
IIT3	$[E * F * \underline{E'}] \xrightarrow{\emptyset} \underline{[E * F * E']}$		

The part that effects the repetition is IIT2b; if this rule was omitted then $[E * F * E']$ would be equivalent to $(E;(F;E'))$.

3.2.7 Recursion

The operational semantics for a variable X with a defining equation $X \stackrel{\mathrm{df}}{=} E$ is given by the inaction rule:

IREC	$\mathcal{C}[X] \xrightarrow{\emptyset} \mathcal{C}[E]$

where $\mathcal{C}[.]$ is a valid PBC syntactic context,[7] such that $\mathcal{C}[X]$ (and so $\mathcal{C}[E]$ as well) is a valid static or dynamic PBC expression. IREC is called the recursion *unfolding* rule. It does not have any derivation rules because, by unfolding, the body E can replace X and, depending on the structure of E, other rules (if any) can be applied. For instance, $[E * F * E'] \approx (E;X)$, where $X \stackrel{\mathrm{df}}{=} (E' \square (F;X))$.

We may now explain why we decided to base the $\cong$-equivalence on lbts^{rdc} instead of lbts. Assume that $\mathcal{X} = \{X\}$ and that the defining equation for the only variable is $X \stackrel{\mathrm{df}}{=} a \| b$. Then $\mathsf{lbts}^{rdc}_{\overline{a\|b}}$ and $\mathsf{lbts}^{rdc}_{\overline{b\|a}}$ are isomorphic transition systems, yet $\mathsf{lbts}_{\overline{a\|b}}$ and $\mathsf{lbts}_{\overline{b\|a}}$ are not since the former has eight nodes and the latter six.

[7] For instance, this may be defined by introducing an extra variable, denoted by ".", so that $\mathcal{C}[.]$ is simply an expression with (possibly) this extra variable; then $\mathcal{C}[expr]$ is obtained by replacing each "." by $expr$.

3.3 Extensions

3.3.1 Generalised iterations

The loop $[E*F*E']$ could be considered as being mainly characterised by its looping part, F, while its initialisation, E, and termination, E', might be seen as being marginal by comparison. In this section, we consider what happens if either of them, or both, are omitted. Consider the following alternative looping constructs:

$[E*F\rangle$	(initialisation E, iteration F, no termination)
$\langle F*E']$	(iteration F, termination E', no initialisation)
$\langle F\rangle$	(iteration F only).

These variants of the loop form a hierarchy: $[E*F\rangle$ could be seen as the same as $E;\langle F\rangle$, the second expression $\langle F*E']$ as the same as $\langle F\rangle;E'$, and the ternary expression $[E*F*E']$ as the same as $E;\langle F*E']$ or $[E*F\rangle;E'$ or $E;\langle F\rangle;E'$.
Modifying the underlying mechanism of the operational rules for the standard iteration, the operational semantics of the alternative loop constructs might be defined by first extending the set of dynamic expressions with: $[G*E\rangle$, $[E*G\rangle$, $\langle G*E]$, $\langle E*G]$ and $\langle G\rangle$, and then adding the derivation rules shown in table 2. While these rules seem to be

IIT11	$\overline{[E*F\rangle} \xrightarrow{\emptyset} [\overline{E}*F\rangle$	IT11	$\dfrac{G \xrightarrow{\Gamma} G'}{[G*F\rangle \xrightarrow{\Gamma} [G'*F\rangle}$
IIT12a	$[\underline{E}*F\rangle \xrightarrow{\emptyset} [E*\overline{F}\rangle$		
IIT12b	$[E*\underline{F}\rangle \xrightarrow{\emptyset} [E*\overline{F}\rangle$	IT12	$\dfrac{G \xrightarrow{\Gamma} G'}{[E*G\rangle \xrightarrow{\Gamma} [E*G'\rangle}$
IIT13	$[E*\underline{F}\rangle \xrightarrow{\emptyset} \underline{[E*F\rangle}$		
IIT21	$\overline{\langle E*F]} \xrightarrow{\emptyset} \langle\overline{E}*F]$	IT21	$\dfrac{G \xrightarrow{\Gamma} G'}{\langle G*F] \xrightarrow{\Gamma} \langle G'*F]}$
IIT22a	$\langle\underline{E}*F] \xrightarrow{\emptyset} \langle\overline{E}*F]$		
IIT22b	$\langle\underline{E}*F] \xrightarrow{\emptyset} \langle E*\overline{F}]$	IT22	$\dfrac{G \xrightarrow{\Gamma} G'}{\langle E*G] \xrightarrow{\Gamma} \langle E*G']}$
IIT23	$\langle E*\underline{F}] \xrightarrow{\emptyset} \underline{\langle E*F]}$		
IIT31	$\overline{\langle E\rangle} \xrightarrow{\emptyset} \langle\overline{E}\rangle$		
IIT32	$\langle\underline{E}\rangle \xrightarrow{\emptyset} \langle\overline{E}\rangle$	IT31	$\dfrac{G \xrightarrow{\Gamma} G'}{\langle G\rangle \xrightarrow{\Gamma} \langle G'\rangle}$
IIT33	$\langle\underline{E}\rangle \xrightarrow{\emptyset} \underline{\langle E\rangle}$		

Table 2: Possible SOS rules for the alternative loop constructs.

derived in a natural way from those given for the ternary iteration operator, they exhibit a surprising (and highly unwanted) behaviour when combined with the choice operator. The following derivations are allowed by IIT21–IIT23 and IT21:

$$\begin{array}{lllllll}
\overline{\langle a * b] \square c} & \xrightarrow{\emptyset} & \overline{\langle a * b]} \square c & \xrightarrow{\emptyset} & \langle \overline{a} * b] \square c & \xrightarrow{\{\{a\}\}} & \langle \underline{a} * b] \square c \\
 & \xrightarrow{\emptyset} & \langle \overline{a} * b] \square c & \xrightarrow{\emptyset} & \overline{\langle a * b]} \square c & \xrightarrow{\emptyset} & \overline{\langle a * b] \square c} \\
 & \xrightarrow{\emptyset} & \langle a * b] \square \overline{c} & \xrightarrow{\{\{c\}\}} & \langle a * b] \square \underline{c} & \xrightarrow{\emptyset} & \underline{\langle a * b] \square c}.
\end{array}$$

That is, it is possible to start performing the loop (by executing one or more a's) and afterwards leave it (without performing the terminal b) and enter the other branch of the choice. A similar example can be given when such loops are nested inside enclosing loops (even if the outer loop is a 'safe' one). In section 4, we shall see that this corresponds to a very specific feature of the Petri net translation, too. This scenario is impossible in the SOS style which is more oriented towards CCS, rather than towards Petri nets, and in which the distinction between the dynamic and static expressions is not made (as discussed in section 2.3). A similarly undesired behaviour may be observed of the first generalised loop construct:

$$\begin{array}{lllllllll}
\overline{[a{*}b\rangle \square c} & \xrightarrow{\emptyset} & [a{*}b\rangle \square \overline{c} & \xrightarrow{\{\{c\}\}} & [a{*}b\rangle \square \underline{c} & \xrightarrow{\emptyset} & \underline{[a{*}b\rangle \square c} & \xrightarrow{\emptyset} & \underline{[a{*}b\rangle} \square c \\
 & \xrightarrow{\emptyset} & [a{*}\underline{b}\rangle \square c & \xrightarrow{\emptyset} & [a{*}\overline{b}\rangle \square c & \xrightarrow{\{\{b\}\}} & [a{*}\underline{b}\rangle \square c. & &
\end{array}$$

Here it is possible to start performing the right branch of the choice and afterward still enter the loop, without performing the initial a. Again, this does not correspond to the kind of behaviour one would expect from a choice structure. The operational semantics of the last generalised loop $\langle E\rangle$ turns out to be the simplest one, but this simplicity is offset by a combination of the problematic features exhibited by the two previous constructs.

The above examples provide justification for the decision to include – in the standard PBC – only the least general iteration operator, which is completely safe when combined in any way with other operators, in whichever semantics. Other loop operators will be used only when it is 'safe'. For instance, $\langle F * E']$ will be used only when it is front-guarded with respect to the enclosing choices and loops. See section 6 for such a use, and [8] for a fuller discussion of various forms of guardedness.

3.3.2 Data expressions

In order to model data variables in a programming language, we shall introduce another generalised iteration. For example, the net corresponding to the declaration $\mathbf{var}\,x : \{0,1\}$, part of which is shown in figure 6, involves transitions of the form $\hat{x}_{vw}$ for all possible values $v,w \in \{0,1\}$. In the formal treatment, instead of introducing another family of operators modelling data, we shall simply add a new family of basic processes. Let us assume that VAR is a set of program variables ranged over by $x,y,z,\ldots$ and that the *token domain* of a variable z is the set $D_z \uplus \{\bullet\}$. The D_z can be thought of as the data domain proper (i.e., the *type*): it is the set introduced by its declaration. The elements of D_z will be called *values* and ranged over by $u,v,\ldots$ The $\bullet$ will be interpreted as an 'undefined value'.

For each program variable z, the set of action particles $\mathrm{A_{PBC}}$ contains all symbols z_{kl} and $\hat{z}_{kl}$ such that $k,l \in D_z \uplus \{\bullet\}$. Moreover, z_{kl} and $\hat{z}_{kl}$ are conjugate action particles,

i.e., $\widehat{z_{kl}} = \hat{z}_{kl}$ and $\widehat{\hat{z}_{kl}} = z_{kl}$. The behaviour of variable z will be represented by a loop-like basic process expression obeying the syntax $[z_{()}]$. Intuitively, it is composed of an initialisation part, a core looping part (but with more complex behaviour than that of the previously defined loop) and a termination part. The intended semantics is that such an expression can first execute $z_{\bullet u}$, for any value u in D_z, then a sequence of zero or more executions of z_{uv}, where $u, v \in D_z$, and terminate (possibly) by an execution of an $z_{u\bullet}$. Each such execution is carried out under a restriction that, if z_{ku} and z_{vl} are two consecutively executed actions, then $u = v$.
For the basic expression $[z_{()}]$ we shall use the following set of rules, which resemble those introduced for the iteration operator:

$$
\boxed{
\begin{array}{lccc}
\text{DAT1} & \overline{[z_{()}]} & \xrightarrow{\{\{z_{\bullet u}\}\}} & [\overline{z_{(u)}}] \\
\text{DAT2} & [\overline{z_{(u)}}] & \xrightarrow{\{\{z_{uv}\}\}} & [\overline{z_{(v)}}] \\
\text{DAT3} & [\overline{z_{(u)}}] & \xrightarrow{\{\{z_{u\bullet}\}\}} & \underline{[z_{(u)}]}
\end{array}
}
$$

In the above, $[\overline{z_{(u)}}]$ is a basic dynamic expression which represents the fact that the program variable z presently has the value u. We illustrate the use of the DAT set of rules using a binary variable z, i.e., one with $D_z = \{0,1\}$:

$$
\begin{array}{llllll}
\overline{[z_{()}]} & \xrightarrow{\{\{z_{\bullet 0}\}\}} & [\overline{z_{(0)}}] \quad (\text{DAT1}) & \xrightarrow{\{\{z_{01}\}\}} & [\overline{z_{(1)}}] & (\text{DAT2}) \\
 & \xrightarrow{\{\{z_{11}\}\}} & [\overline{z_{(1)}}] \quad (\text{DAT2}) & \xrightarrow{\{\{z_{1\bullet}\}\}} & \underline{[z_{()}]} & (\text{DAT3})
\end{array}
$$

Notice how in this derivation the expression changes according to the value being propagated; moreover, at the end, the same expression is obtained as that at the start (except for the overbar and underbar, respectively). The full behaviour of a program variable z may then be modelled by $([z_{()}] \Box \{z_{\bullet\bullet}\})$, in order to represent, through $z_{\bullet\bullet}$, moreover the fact that the variable may be destroyed before any true access to it is performed.

3.3.3 Generalised operators

We extend the three binary control flow operators by allowing a variable number of arguments indexed by some nonempty countable indexing set I:

$$\|_{i\in I} E_i \;\; , \;\; \Box_{i\in I} E_i \;\; , \;\; ;_{i\in I} E_i \, . \tag{10}$$

Such a notation is motivated by the associativity of the corresponding binary operators. For the indexed sequence operator, I must be a finite or infinite sequence; for the indexed choice and parallel composition operators, I need not be ordered since the corresponding binary operators are also commutative. When I is finite, the expressions in (10) are equivalent[8] to standard expressions; for instance, $\|_{i\in\{0,1,2\}} E_i$ is equivalent to $E_0 \| (E_1 \| E_2)$. When I is infinite, we can provide equivalent definitions in terms of a system of recursive equations. If we assume that I is the set of natural numbers then the three constructs in (10) are equivalent to the variable X_0 which is evaluated in the

[8] In the sense of the $\cong$-relation, here and later in this paragraph.

context of an infinite set of recursive equations, for every $i \geq 0$, defined by, respectively, $X_i \stackrel{\mathrm{df}}{=} E_i \| X_{i+1}$, $X_i \stackrel{\mathrm{df}}{=} E_i \,\square\, X_{i+1}$ and $X_i \stackrel{\mathrm{df}}{=} E_i ; X_{i+1}$.
As suggested by the properties of commutativity and idempotence of the synchronisation, restriction and scoping operators, one may introduce operators such as $E \;\mathsf{sy}\; A$, $E \;\mathsf{rs}\; A$ and $[A : E]$, where $A \subseteq \mathsf{A}_{\mathrm{PBC}}$ is a set of action particles. The idea here is to apply the corresponding unary operations for all the action particles in A, in any order (due to commutativity) and without worrying about repetitions (idempotence leads to the observation that considering multisets instead of sets of action particles would add nothing in that respect). For finite sets A, this simply may be a way of compacting the notation, but with infinite sets the expressiveness of the model is strictly increased.

3.3.4 Extended PBC syntax

Having introduced the basic PBC and discussed a number of its possible extensions, we will now give the syntax for an extended version of the basic PBC, which incorporates some of the extensions that we have mentioned:

$$\begin{array}{lll} E & ::= & \alpha \mid X \mid E \| E \mid E \,\square\, E \mid E ; E \mid [E * E * E] \mid [z_{()}] \mid \\ & & E[f] \mid E \;\mathsf{sy}\; a \mid E \;\mathsf{rs}\; a \mid [a : E] \mid E \;\mathsf{sy}\; A \mid E \;\mathsf{rs}\; A \mid [A : E], \end{array} \tag{11}$$

where $A \subseteq \mathsf{A}_{\mathrm{PBC}}$ and z is a program variable. The meaning of the remaining items is the same as in (3). The *dynamic (extended) PBC expressions* are defined by the following syntax, where E is a static PBC expression given by (11):

$$\begin{array}{lll} G & ::= & \overline{E} \mid \underline{E} \mid G \| G \mid G \,\square\, E \mid E \,\square\, G \mid G ; E \mid E ; G \mid \\ & & [G * E * E] \mid [E * G * E] \mid [E * E * G] \mid [\overline{z_{(u)}}] \mid \\ & & G[f] \mid G \;\mathsf{sy}\; a \mid G \;\mathsf{rs}\; a \mid [a : G] \mid G \;\mathsf{sy}\; A \mid G \;\mathsf{rs}\; A \mid [A : G], \end{array} \tag{12}$$

where $A \subseteq \mathsf{A}_{\mathrm{PBC}}$, z is a program variable, and $u \in D_z$.

4 Petri Net Semantics

Petri nets have long been provided with various (coherent) behavioural semantics (e.g., [34, 40]), in particular concurrent semantics such as trace semantics [29], step semantics [20], process semantics [5, 23], and partial word semantics [24, 42, 45]. Hence, a natural idea to get a fully fledged (concurrent) semantics for a process algebra is to associate a net to each expression of the algebra. This technique has already been exploited for various existing process algebras [11, 13, 21, 22, 35, 43], but in many cases only fragments of the theory have been successfully translated. Here we shall describe how to do the job not only in all generality, but also fully compositionally, due to the careful choice of the operators of the PBC and very general mechanisms introduced to combine nets. Indeed, in order to get a compositional way of translating (dynamic as well as static) PBC expressions into nets we need to define for them at least the same operators as for the process algebra. Then the translation will simply be a homomorphism. The present section shows how to do this for the recursion-free PBC. Our approach to compositionality will be based on transition refinement, i.e., each operator on nets, op, will be based on a finite net Ω_{op} whose transitions $t_1, t_2, \ldots, t_n$ are refined by the corresponding nets $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ in the process of forming a new net

$op(\Sigma_1, \Sigma_2, \ldots, \Sigma_n) = \Omega_{op}(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$. To carry this out, we need to be able to distinguish those (labelled) nets that are easily composable with one another, from the rest which are not. These considerations will be contained in sections 4.1 and 4.2 of the present section, and they will immediately be applied to the basic PBC, in section 4.3.

4.1 Labelled nets and boxes

We delineate several classes of labelled Petri nets whose interfaces are expressed by labellings of places and transitions, collectively called boxes because they can be viewed as nets with an interface. There are two main classes of boxes which we will be interested in, viz. plain boxes and operator boxes.
Plain boxes – defined in section 4.1.4 – are the basic semantical objects of interest. They form the class of elements of our Petri net domain upon which various operators are defined, just as expressions form the domain of a process algebra upon which process algebraic operators are defined. When giving the Petri net semantics of a process algebra (such as PBC or CCS), we will associate a plain box with every expression.
Operator boxes – defined in section 4.2.1 – are patterns (or functions) defining the ways of constructing new plain boxes out of given ones. When translating a process algebra into Petri nets, we aim at associating a specific operator box with every operator of the process algebra. It is one of the characteristic features of our approach that the same type of nets – boxes – serve to describe two seemingly very different objects, namely the elements and the operators of the semantical domain. However, this is very similar to viewing constants as nullary functions (in logics, for instance).
We introduce the kind of nets we shall need using a simple example. A possible model for the dynamic expression $\overline{\alpha; (\alpha \| \alpha)}$ might look like the net depicted in figure 8. Note that this net is safe under the marking shown there.

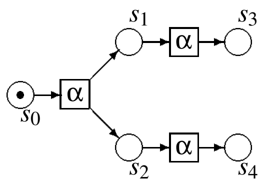


Figure 8: Net of $\overline{\alpha; (\alpha \| \alpha)}$.

Clearly, the action name α cannot be used to identify the transitions, since there are three of them, all corresponding to the same action α. As a result, to model static or dynamic expressions we need to employ Petri nets with transitions being labelled by action names. For Petri nets defining net operators, we shall also use transition labelling, but with labels corresponding to general relabellings, in order to allow combining together transitions coming from the composed nets. And, since action names may be treated as a special kind of (constant) relabellings, the latter may be used in full generality.
In the above net, moreover, three different kinds of places can be identified. The place s_0 is special in the sense that it contains the token corresponding to the expression in its initial state. And, by symmetry, the places s_3 and s_4, when holding one token each, characterise the terminal state of the expressions. The two remaining places, s_1 and s_2,

may be considered as internal and contributing to intermediate markings corresponding to intermediate dynamic expressions. The different role of the places may be captured by a suitable labelling mechanism, with three possible values corresponding to the three kinds of places.

4.1.1 Actions and relabellings

We assume a set Lab of *actions* to be given. At this point Lab is an arbitrary set, but later we shall consider Lab to be the structured set $\mathsf{Lab}_{\mathrm{PBC}}$ of actions used in the PBC expressions. The intuition behind an element $\alpha \in \mathsf{Lab}$ is that α expresses some interface activity. The notation 'Lab' has been chosen for the action set because actions will serve as transition labels. When executing a transition, or a set of concurrently enabled transitions, we shall then be able to consider the corresponding action or (multiset of) actions. A *relabelling* ρ is a relation

$$\rho \subseteq (\mathsf{mult}(\mathsf{Lab})) \times \mathsf{Lab} \tag{13}$$

such that $(\emptyset, \alpha) \in \rho$ if and only if $\rho = \{(\emptyset, \alpha)\}$. The intuition behind a pair (Γ, α) belonging to ρ is that it specifies some interface change which can be applied to a (finite) group of transitions whose labels match the argument, i.e., the multiset of actions Γ. Since Γ, being a multiset, is an unordered object, the order of the transitions in such a group does not matter, and we immediately obtain a kind of simple commutativity of the operation described by ρ.

Three specific relabellings are of particular interest. A *constant* relabelling, $\rho_\alpha = \{(\emptyset, \alpha)\}$, where α is an action in Lab, can be identified with α itself, so that we may consider the set of actions Lab to be embedded in the set of all relabellings. If a relabelling is not constant, then it will be called *transformational*; in that case, the empty set will not be in its domain, in order not to *create an action out of nothing*.

The *restriction* $\rho_{\mathsf{Lab}'} = \{(\{\alpha\}, \alpha) \mid \alpha \in \mathsf{Lab}'\}$ only keeps the actions belonging to some set $\mathsf{Lab}' \subseteq \mathsf{Lab}$.

The *identity* relabelling, $\rho_{id} = \{(\{\alpha\}, \alpha) \mid \alpha \in \mathsf{Lab}\}$ captures the 'keep things as they are' interface (non)change; it is a special restriction: $\rho_{id} = \rho_{\mathsf{Lab}}$.

4.1.2 Labelled nets

In this paper, by a *(marked) labelled net* we will mean a tuple

$$\Sigma = (S, T, W, \lambda, M) \tag{14}$$

such that: S and T are disjoint sets of respectively *places* and *transitions*; W is a *weight function* from the set $(S \times T) \cup (T \times S)$ to the set of natural numbers $\mathbf{N}$; λ is a *labelling* function for places and transitions such that $\lambda(s) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$, for every place $s \in S$, and $\lambda(t)$ is a relabelling ρ of the form (13), for every transition $t \in T$; and M is a *marking*, i.e., a mapping assigning a natural number to each place $s \in S$. This generalises the net model we considered in section 2.2. We adopt the standard rules about representing nets as directed graphs, viz. places are represented as circles, transitions as rectangles, the flow relation generated by W is indicated by arcs annotated with the corresponding weights, and markings are shown by placing tokens within circles. As usual, the zero weight arcs will be omitted and the unit weight arcs (or unitary arcs) will be left as plain arcs, i.e., unannotated. To avoid ambiguity, we will sometime decorate the various components of Σ with the index Σ; thus, T_Σ denotes the set of transitions of Σ, etc. A net is *finite* if both S and T are finite sets.

Σ_0

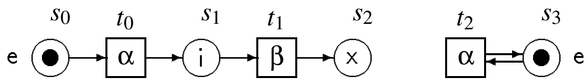


Figure 9: A labelled net, Σ_0.

Figure 9 shows the graph of a labelled net $\Sigma_0 = (S_0, T_0, W_0, \lambda_0, M_0)$ defined thus:

$$\begin{aligned} S_0 &= \{s_0, s_1, s_2, s_3\} \quad \text{and} \quad T_0 \;=\; \{t_0, t_1, t_2\} \\ W_0 &= ((TS \cup ST) \times \{1\}) \cup (((S_0 \times T_0) \setminus ST) \times \{0\}) \cup (((T_0 \times S_0) \setminus TS) \times \{0\}) \\ &\quad \text{where } TS = \{(t_0, s_1), (t_1, s_2), (t_2, s_3)\} \text{ and } ST = \{(s_0, t_0), (s_1, t_1), (s_3, t_2)\} \\ \lambda_0 &= \{(s_0, \mathsf{e}), (s_1, \mathsf{i}), (s_2, \mathsf{x}), (s_3, \mathsf{e}), (t_0, \alpha), (t_1, \beta), (t_2, \alpha)\} \\ M_0 &= \{(s_0, 1), (s_1, 0), (s_2, 0), (s_3, 1)\}. \end{aligned}$$

We adopt finite step sequence semantics for a labelled net $\Sigma = (S, T, W, \lambda, M)$, in order to capture the potential concurrency in the behaviour of the system modelled by Σ. A finite multiset of transitions U, called a *step*, is *enabled* by Σ if for every place $s \in S$,

$$M(s) \;\geq\; \sum_{t \in U} (U(t) \cdot W(s,t)).$$

We denote this by $\Sigma\,[U\rangle$, or by $M\,[U\rangle$ if the underlying net is understood. An enabled step U can be *executed* leading to a follower marking M' defined, for every place $s \in S$, by

$$M'(s) \;=\; M(s) + \sum_{t \in U} (U(t) \cdot (W(t,s) - W(s,t))).$$

We will denote this by $M\,[U\rangle\,M'$ or $\Sigma\,[U\rangle\,\Theta$, where Θ is the labelled net (S, T, W, λ, M'). Transition labelling may be extended to steps, through the formula

$$\lambda(U) \;=\; \sum_{t \in U} (U(t) \cdot \{\lambda(t)\}) \;\in\; \mathsf{mult}(\mathsf{Lab}).$$

A net exhibits *auto-concurrency* if some follower marking enables a step which is not a set.Notice that the label of a step may need to be a multiset rather than a set, even if the net exhibits no auto-concurrency, since different concurrent transitions may have the same label (as in figure 8). Although we use the same term 'step' to refer to both a finite set of transitions here, and a finite multiset of actions in section 3, it will always be clear from the context which one is meant. The notation for action based steps will be $\Sigma\,[\Gamma\rangle_{\mathsf{lab}}\,\Theta$, etc.

A *finite step sequence* of Σ is a finite (possibly empty) sequence $\sigma = U_1 \ldots U_k$ of finite multisets of transitions for which there are labelled nets $\Sigma_0, \ldots, \Sigma_k$ such that $\Sigma = \Sigma_0$ and for every $1 \leq i \leq k$, $\Sigma_{i-1}\,[U_i\rangle\,\Sigma_i$. Depending on the context, this shall be denoted by one of the following notations:

$$\Sigma\,[\sigma\rangle\,\Sigma_k \quad , \quad M_\Sigma\,[\sigma\rangle\,M_{\Sigma_k} \quad , \quad \Sigma_k \in [\Sigma\rangle \quad \text{or} \quad M_{\Sigma_k} \in [M_\Sigma\rangle\,.$$

Moreover, the marking M_{Σ_k} will be called *reachable* from M_Σ, and Σ_k *derivable* from Σ. The empty step will always be enabled, but it can be ignored when one considers a step sequence. The only difference is that here the empty step only relates a net to itself; i.e., $\Sigma\,[\emptyset\rangle\,\Theta \;\Leftrightarrow\; \Sigma = \Theta \;\Leftrightarrow\; \Sigma\,[\varepsilon\rangle\,\Theta$, where ε denotes the empty sequence.

Consider the marked labelled net Σ_0 shown in figure 9. There, transitions t_0 and t_2 are enabled concurrently and, hence, $\{t_0,t_2\}$ is an enabled step. After this step has been executed, transitions t_1 and t_2 are enabled concurrently and, hence, $\{t_0,t_2\}\{t_1,t_2\}$ is a step sequence of the net from its shown marking. A step does not need to be maximal. Thus, for instance, $\{t_0\}$ is also a step of Σ_0, and $\{t_0\}\{t_1\}\{t_2\}\{t_2\}$ and $\{t_0,t_2\}\{t_1\}\{t_2\}$ are step sequences. In terms of labelled steps, the step sequence $\{t_0,t_2\}\{t_1\}\{t_2\}$ corresponds to $\{\alpha,\alpha\}\{\beta\}\{\alpha\}$. Notice that different step sequences may correspond to the same labelled step sequence. For example, both $\{t_0\}\{t_2\}\{t_1\}$ and $\{t_2\}\{t_0\}\{t_1\}$ correspond to $\{\alpha\}\{\alpha\}\{\beta\}$.

If the labelling of a place s in a labelled net Σ is e then s is an *entry* place, if i then s is an *internal* place, and if x then s is an *exit* place. By convention, ${}^{\circ}\Sigma$, Σ° and $\ddot{\Sigma}$ denote respectively the entry, exit and internal places of Σ. For every place (transition) x, we use ${}^{\bullet}x$ to denote is pre-set, i.e., the set of all transitions (places) y such that there is an arc from y to x, that is, $W(y,x)>0$. The post-set $x^{\bullet}$ is defined in a similar way. The pre- and post-set notation extends in the usual way to sets R of places and transitions, e.g., ${}^{\bullet}R=\bigcup\{{}^{\bullet}r \mid r\in R\}$. In what follows, all nets are assumed to be T-*restricted*, i.e., the pre- and post-sets of each transition are nonempty. No assumption of that kind is made for places. For the labelled net of figure 9 we have ${}^{\circ}\Sigma_0=\{s_0,s_3\}$, $\Sigma_0^{\circ}=\{s_2\}$, ${}^{\bullet}s_0=\emptyset$, $s_0^{\bullet}=\{t_0\}$ and $\{s_0,s_1\}^{\bullet}=\{t_0,t_1\}={}^{\bullet}\{s_1,s_2\}$.

The labelled net Σ is *simple* if W always returns 0 or 1 and *pure* if for all transitions $t\in T$, ${}^{\bullet}t\cap t^{\bullet}=\emptyset$. If Σ is not pure then there are $s\in S$ and $t\in T$ such that $(W(s,t)\cdot W(t,s))>0$. In such a case, the pair $\{s,t\}$ will be called a *side-loop*, and s a *side-condition* of t. Thus, being pure amounts to side-loop freeness, or side-condition freeness. The net in figure 9 is finite and simple, but is not pure as it contains a side-loop, $\{s_3,t_2\}$.

The marking M of Σ is *safe* if for all $s\in S$, $M(s)\in\{0,1\}$. A safe marking can and will often be identified with the set of places to which it assigns 1. A safe marking is *clean* if it is not a proper superset of ${}^{\circ}\Sigma$ nor Σ°, i.e., if ${}^{\circ}\Sigma\subseteq M$ or $\Sigma^{\circ}\subseteq M$ implies ${}^{\circ}\Sigma=M$ or $\Sigma^{\circ}=M$, respectively. The marking of the net in figure 9 is both safe and clean. It would cease to be clean if we added a token to it, even if this new token would be put on one of the entry places (because a clean marking must also be safe). A marked net is called *safe* (*clean*) if all its reachable markings are safe (resp., clean), *k-bounded* if no reachable marking puts more than k tokens on any place (so that 1-boundedness is the same as safeness), and *bounded* if there is some k such that it is k-bounded.

A symmetric (and, by T-restrictedness, irreflexive), not necessarily transitive, relation ind_{Σ} is defined on the transition set of a labelled net Σ, by

$$\mathsf{ind}_{\Sigma} \;=\; \{(t,u)\in T\times T \mid ({}^{\bullet}t\cup t^{\bullet})\cap({}^{\bullet}u\cup u^{\bullet})=\emptyset\}.$$

This relation is called the *independence relation*, because two distinct transitions belonging to ind_{Σ} have no impact on their respective environments. If they are both enabled individually, then they are enabled simultaneously. Conversely, if Σ is safe, it can be shown that, whenever two transitions occur in the same step, then they are independent.

We will use three explicit ways of modifying the marking of $\Sigma=(S,T,W,\lambda,M_{\Sigma})$. We define $\lfloor\Sigma\rfloor$ as $(S,T,W,\lambda,\emptyset)$; typically, this operation is used when $M_{\Sigma}\neq\emptyset$, since it corresponds to erasing all the tokens. Moreover, we define $\overline{\Sigma}$ and $\underline{\Sigma}$ as, respectively, $(S,T,W,\lambda,{}^{\circ}\Sigma)$ and $(S,T,W,\lambda,\Sigma^{\circ})$. These operations are typically applied if $M_{\Sigma} = \emptyset$, and they correspond to placing one token on each entry place (respectively, one token on each exit place). We will call ${}^{\circ}\Sigma$ the *entry marking*, and Σ° the *exit marking* of Σ; note that both are safe and clean. Note also that $\lfloor . \rfloor$, $\overline{(.)}$ and $\underline{(.)}$ are syntactic operations having nothing to do with derivability (reachability) in the sense of the step sequence semantics defined above.

4.1.3 Equivalence notions

As for PBC expressions, various behavioural equivalence or congruence notions may be defined for labelled nets. It may first be observed that the whole set of step sequences may be specified by defining the *full reachability graph* of a net, whose nodes are all the reachable markings (or equivalently, the reachable marked nets) and whose arcs are labelled with steps which transform one marking into another. For example, figure 10 represents the full reachability graph of the labelled net Σ_0 shown in figure 9; the empty steps are left implicit; and the circled dot indicates the initial node, M_{Σ_0}. The arc labels may be transition steps (as in figure 10) or labelled steps. Any finite path in this graph starting at the initial node specifies a legal step sequence of the marked net Σ_0, and vice versa.

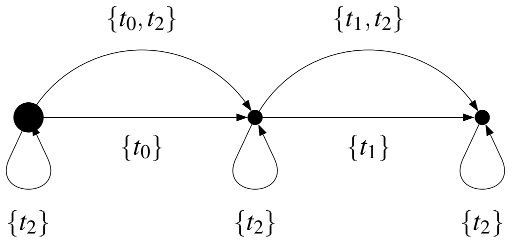


Figure 10: Full reachability graph of Σ_0 (figure 9).

Using the reachability graph to represent the overall behaviour of a labelled net Σ, leads to the same kind of difficulties as encountered in section 3.2.1 when we discussed the operational semantics of PBC expressions. That is, the isomorphism of reachability graphs is not preserved by, e.g., choice composition of nets (even if we consider transition labels), as illustrated in figure 11. We address this problem by introducing a device similar to that applied in the case of PBC expressions. Recall that the idea there was to augment the behaviour with two auxiliary moves, skip and redo, which could transform an expression in the initial state into an expression in the terminal state, and vice versa. In the case of the labelled net Σ, we achieve a similar effect by adding to it (artificially) two fresh transitions, skip and redo, so that ${}^{\bullet}\mathsf{skip} = \mathsf{redo}^{\bullet} = {}^{\circ}\Sigma$, $\mathsf{skip}^{\bullet} = {}^{\bullet}\mathsf{redo} = \Sigma^{\circ}$, $\lambda(\mathsf{skip}) = \mathsf{skip}$ and $\lambda(\mathsf{redo}) = \mathsf{redo}$. Moreover, we assume that all the arcs adjacent to the skip and redo transitions are unitary, $\mathsf{redo},\mathsf{skip} \notin \mathsf{Lab}$, and ${}^{\circ}\Sigma \neq \emptyset \neq \Sigma^{\circ}$. The latter condition is needed to ensure that the two new transitions are T-restricted. Denote the net Σ augmented with skip and redo by Σ_{sr}. Then the labelled transition system gener-

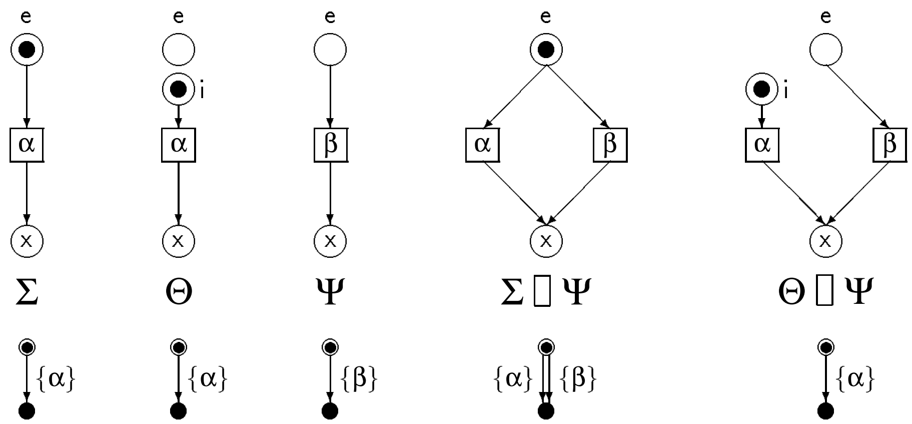


Figure 11: Reachability graph isomorphism is not preserved by choice.

ated by Σ is defined as $\mathsf{lbts}_\Sigma = (V, L, A, v_0)$ where $V = \{\Theta \mid \Theta_{\mathsf{sr}} \in [\Sigma_{\mathsf{sr}}\rangle\ \}$ is the set of states, $v_0 = \Sigma$ is the initial state, $L = \mathsf{mult}(\mathsf{Lab} \cup \{\mathsf{redo}, \mathsf{skip}\})$ is the set of arc labels, and the arcs are given by:

$$A \ = \ \{(\Theta, \Gamma, \Psi) \mid \Theta_{\mathsf{sr}} \in [\Sigma_{\mathsf{sr}}\rangle \ \wedge \ \Theta_{\mathsf{sr}}\, [\Gamma\rangle_{\mathsf{lab}}\, \Psi_{\mathsf{sr}}\}.$$

In other words, lbts_Σ is the labelled reachability graph of Σ_{sr} with all the references to skip and redo in the nodes (but not on the arcs) of the graph erased. The labelled transition system generated by an unmarked labelled net Σ, corresponding to a marked net with the empty marking, is defined as $\mathsf{lbts}_\Sigma = \mathsf{lbts}_{\overline{\Sigma}}$.

Figure 12 shows how augmenting labelled nets with the redo and skip transitions allows one to discriminate between the nets Σ and Θ depicted in figure 11. Thus, skip and redo allow for distinguishing the entry and exit states from the other ones, and for modelling the fact that if a net is left (through the exit state), it may later be possible to re-enter it (through the entry state).

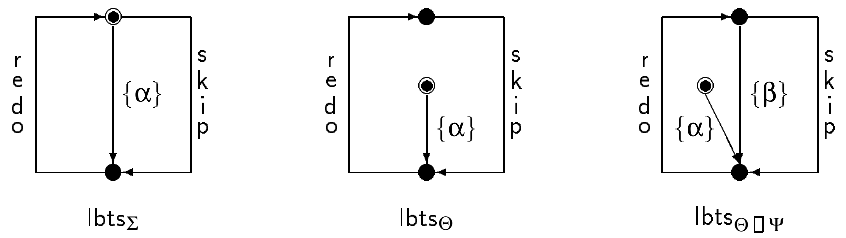


Figure 12: Discriminating lbts's.

Two labelled nets, Σ and Θ, will be called *lbts-isomorphic*, denoted $\Sigma \cong \Theta$, if lbts_Σ and lbts_Θ are isomorphic transition systems, and *strongly equivalent*, denoted $\Sigma \approx \Theta$, if lbts_Σ and lbts_Θ are strongly equivalent transition systems. Notice that there is no need to consider an lbts-isomorphism up to the empty moves, as in section 3.2.1, since here $\Sigma\,[\emptyset\rangle\,\Sigma'$ iff $\Sigma = \Sigma'$. Figure 13 shows an example of strongly equivalent (but not lbts-isomorphic) nets. Note that adding or dropping dead transitions (i.e., transitions which may never occur in the augmented net) preserves $\cong$-equivalence and $\approx$-equivalence.

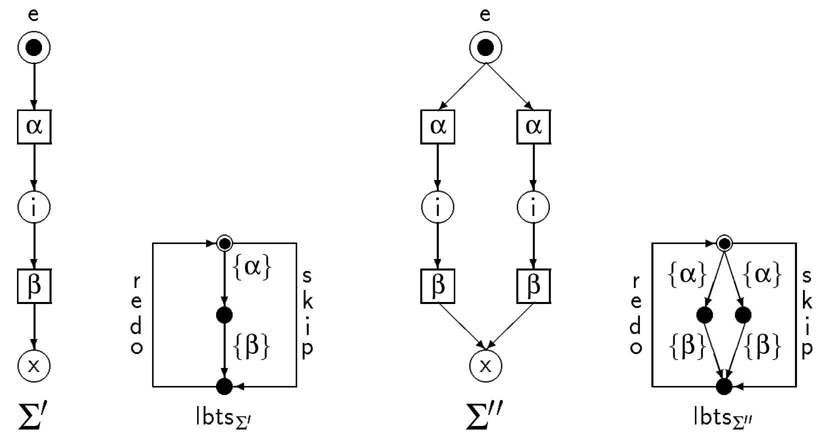


Figure 13: Two strongly equivalent labelled nets.

What we have done above is not the only way of defining relevant equivalence notions in the domain of labelled nets. In particular, because lbts-isomorphism and strong equivalence are essentially behavioural, it may be difficult to check if two nets are so related. Since we are working now with nets, it is also possible to define equivalence notions based on the structure of nets rather than on their behaviour (of course, a structural equivalence should imply a behavioural one). But, again, there are different ways to do so.

The strongest structural equivalence, other than equality, is net isomorphism. Two labelled nets, Σ and Θ, are *isomorphic*, denoted $\Sigma\ \mathsf{iso}\ \Theta$, if their graphs are isomorphic, i.e., if there is a bijective sort-preserving mapping $\mathsf{iso}\colon S_\Sigma \cup T_\Sigma \to S_\Theta \cup T_\Theta$ such that for every $x \in S_\Sigma \cup T_\Sigma$, $\lambda_\Theta(\mathsf{iso}(x)) = \lambda_\Sigma(x)$, for every $s \in S_\Sigma$, $M_\Theta(\mathsf{iso}(s)) = M_\Sigma(s)$, and for all $s \in S_\Sigma$ and $t \in T_\Sigma$, $W_\Theta(\mathsf{iso}(s), \mathsf{iso}(t)) = W_\Sigma(s,t)$ and $W_\Theta(\mathsf{iso}(t), \mathsf{iso}(s)) = W_\Sigma(t,s)$.

Weaker equivalences are obtained by allowing nets differ only by duplicating places and transitions. Two places s and s' duplicate each other in a labelled net Σ if $\lambda_\Sigma(s) = \lambda_\Sigma(s')$, $M_\Sigma(s) = M_\Sigma(s')$, and for every transition t, $W_\Sigma(s,t) = W_\Sigma(s',t)$ and $W_\Sigma(t,s) = W_\Sigma(t,s')$; then, in any evolution of the net, the two places behave in an identical way and do not add anything with respect to each other. Similarly, two transitions t and t' duplicate each other if $\lambda_\Sigma(t) = \lambda_\Sigma(t')$, and for every place s, $W_\Sigma(s,t) = W_\Sigma(s,t')$ and $W_\Sigma(t,s) = W_\Sigma(t',s)$; then, in any evolution of the net, the two transitions lead to the same labelled steps and do not add anything with respect to each other. Clearly, the *duplicating* relation is an equivalence between the places and between the transitions, and it is possible to replace in the net the place/transition set by the set of their equivalence classes, thus obtaining a net with essentially the same behaviour. More precisely, two labelled nets, Σ and Θ, will be called *place-duplicating* (*transition-duplicating*, or *node-duplicating*, respectively), denoted $\Sigma\ \mathsf{iso}_S\ \Theta$ ($\Sigma\ \mathsf{iso}_T\ \Theta$ or $\Sigma\ \mathsf{iso}_{ST}\ \Theta$, respectively), if they lead to isomorphic nets when their places (transitions or nodes, respectively) are replaced by their duplicating equivalence class. It is a straightforward observation that in the domain of labelled nets,

$$\mathsf{iso} \subset \mathsf{iso}_S \subset \mathsf{iso}_{ST} \subset \cong \subset \approx$$

$$\text{and} \quad \mathsf{iso} \subset \mathsf{iso}_T \subset \mathsf{iso}_{ST} \subset \cong \subset \approx .$$

The four notions of structural equivalence are illustrated in figure 14. In particular, we have Σ_1 iso_T Σ_2 iso_{ST} Σ_3 iso_S Σ_1, but Σ_4 is not equivalent to any of other three nets.

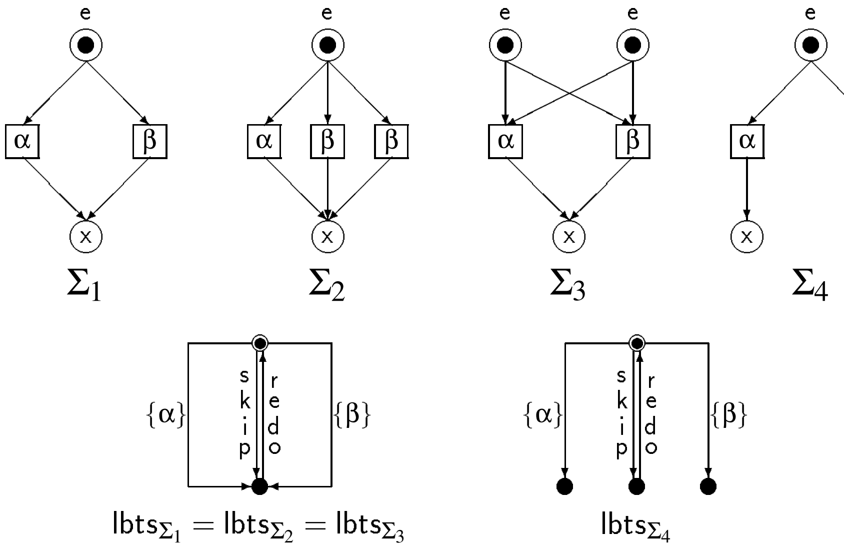


Figure 14: Four labelled nets and their lbts's.

4.1.4 Boxes and (static or dynamic) plain boxes

A *box* is a labelled net Σ such that there is at least one entry and at least one exit place (Σ is ex-*restricted*) and, moreover, the entry places are free from incoming arcs and the exit places are free from outgoing arcs (Σ is *ex-directed*), that is, if ${}^\circ\Sigma \neq \emptyset \neq \Sigma^\circ$ and ${}^\bullet({}^\circ\Sigma) = \emptyset = (\Sigma^\circ)^\bullet$. We require Σ to be ex-restricted to ensure that the operation of net refinement is well defined. We will see later that, and how, the theory depends on this assumption, cf. section 4.2.4. This property ensures, moreover, that lbts_Σ is well defined for a box Σ. The reason for demanding ex-directedness is more subtle and is motivated by our intention to obtain certain behavioural properties of compositionally defined boxes. We return to this issue after discussing net refinement, in section 4.2.4. Note that the labelled net Σ_0 in figure 9 is not a box since it is not ex-directed.

A box Σ is, by definition, *plain* if for each transition $t \in T_\Sigma$, the label $\lambda_\Sigma(t)$ is a constant relabelling. Hence, by our convention, every transition label in a plain box is an action in Lab.

We now define some important behavioural properties of plain boxes. A plain box Σ will be called *static* if its marking M_Σ is empty and all the markings reachable from its entry marking ${}^\circ\Sigma$ are safe and clean. Static boxes are our analogue of static expressions (i.e., expressions without under- or over-barring). The safeness and cleanness conditions delineate the class of nets which we will be considering. Later, we will show that all boxes that will be constructed for process algebraic expressions (or for concurrent programs, for that matter) will automatically satisfy these properties.

A plain box Σ is *dynamic* if its marking M_Σ is nonempty, $\lfloor\Sigma\rfloor$ is a static box, and all the markings reachable from M_Σ are safe and clean. Dynamic boxes correspond to dynamic expressions, i.e., expressions for which active subexpressions are indicated by overbarring and underbarring. Note that if Σ is a static box and Θ is derivable from $\overline{\Sigma}$ then Θ is a dynamic box; in particular, $\overline{\Sigma}$ itself is a dynamic box. However, in this definition, it is not required that M_Σ can be reached from ${}^\circ\Sigma$.

Typically, we shall be interested in dynamic boxes Σ which are reachable from $\overline{\Sigma}$, but when combining reachable dynamic boxes together, it may sometimes happen that the result is not reachable, much in the same way as some syntactically valid dynamic expressions may not be reached from their corresponding initial expression (for instance, $(a;\overline{b})\ \mathsf{rs}\ a$ is not reachable from $\overline{(a;b)\ \mathsf{rs}\ a}$).

When a box Σ is started from a nonempty marking (in particular, ${}^\circ\Sigma$, because of ex-restrictedness), its reachable markings are always nonempty (because of T-restrictedness). On the other hand, the empty marking of a box has no successor markings except itself (reachable by the empty step sequence). Thus, the distinction between static and dynamic boxes is invariant over behaviour. Moreover, it may be noticed that the notions of being static or dynamic, and their invariance over behaviour, do not rely on ex-directedness; hence they may be extended to boxes augmented with the $\mathsf{redo}/\mathsf{skip}$ transitions.

Within the set of dynamic boxes we further distinguish two special classes, called *entry* and *exit* boxes, which comprise all dynamic boxes Σ such that M_Σ is, respectively, ${}^\circ\Sigma$ and Σ°. Note that the sets of static boxes, entry boxes and exit boxes, are in bijection with each other: the functions associating with a static box Σ the entry box $\overline{\Sigma}$ and the exit box $\underline{\Sigma}$ are bijections from the set of static boxes to the set of entry boxes and to the set of exit boxes, respectively. The set of dynamic boxes is much larger: it properly contains the set of entry boxes and the set of exit boxes, but is disjoint with the set of static boxes.

The families of plain static, dynamic, entry and exit boxes will, respectively, be denoted by Box^s, Box^d, Box^e and Box^x, and the whole family of plain boxes will be denoted by Box.

Proposition 4.1 Let Σ be a dynamic box and U be a step enabled by Σ. Then every net derivable from Σ is a dynamic box; U is a set of mutually independent transitions, $U \times U \subseteq \mathsf{ind}_\Sigma \cup id_{T_\Sigma}$; and all the arcs adjacent to the transitions in U are unitary, i.e., $W_\Sigma((U \times S_\Sigma) \cup (S_\Sigma \times U)) \subseteq \{0,1\}$. □

Proposition 4.2 A box Σ is dynamic (static) if and only if so is Σ_{sr}. □

Two behavioural conditions were imposed on the markings M reachable from the entry marking, $M \in [{}^\circ\Sigma\rangle$, of a static box Σ. First, we require M to be safe in order to ensure that the semantics of the boxes is as simple as possible and, in particular, that the nets we consider do not allow auto-concurrency, and that one can directly use a partial order semantics of Petri nets in the style of Mazurkiewicz [29], as described in [28]. The second condition, that M is a clean marking, is a consequence of the first condition and our wish to use iterative constructs in the algebra of nets. Moreover, it may be observed that when a box is augmented with the two $\mathsf{redo}/\mathsf{skip}$ transitions, cleanness implies that if a redo is performed, then we get the entry marking ${}^\circ\Sigma$ and the net remains safe (and clean); on the other hand, if $\overline{\Sigma}$ were not clean, $\overline{\Sigma}_{\mathsf{sr}}$ would not be safe either (nor clean). Although static boxes are our primary interest, we also need to be able to represent intermediate markings by dynamic boxes. The latter, more generally, will allow us to exhibit a very close relationship between the operational and Petri net semantics of process expressions.

4.2 Net refinement

The basis for providing plain boxes with an algebraic structure, and hence for defining a compositional denotational semantics of static and dynamic PBC expressions, will be a general *simultaneous refinement and relabelling meta-operator* (*net refinement*, for short). It captures a mechanism by which transition refinement and interface change defined by relabellings are combined. Both operations are defined for an arbitrary simple box Ω which serves as a pattern for gluing together a tuple of plain boxes $\vec{\Sigma}$ (one plain box for every transition in Ω) along their entry and exit interfaces. The relabellings annotating the transitions of Ω specify the interface changes to which the boxes in $\vec{\Sigma}$ are subjected.

4.2.1 Operator boxes

An *operator box* is a simple finite box Ω whose relabellings are transformational, i.e., non-constant. We will assume that $T_\Omega = \{v_1, \ldots, v_n\}$ is an arbitrary but fixed ordering of the transitions of Ω. Let $\vec{\Sigma} = (\Sigma_{v_1}, \ldots, \Sigma_{v_n})$ be a tuple of plain boxes; for every $v \in T_\Omega$, $\Sigma_v = (S_v, T_v, W_v, \lambda_v, M_v)$. We will refer to $\vec{\Sigma}$ as an Ω-*tuple*, and treat it as a vector of plain boxes. We shall not require that the boxes in $\vec{\Sigma}$ be distinct.

As the transitions of an operator box are meant to be refined, i.e., replaced, by full systems represented by the boxes Σ_v, it seems reasonable to consider that their execution may take some (arbitrarily long) time or, indeed, may last indefinitely (if the subsystem represented by Σ_v deadlocks or works endlessly). This may be captured by a special kind of extended markings. A *complex marking* of an operator box Ω is a pair (M,Q) composed of a standard marking M of Ω and a finite multiset Q of ('engaged') transitions of Ω. M may be considered as the *real* part, and Q as the *imaginary* part of the complex marking. A standard marking M may then be identified with the complex marking $(M,\emptyset)$. The enabling and execution rules are extended thus. Let U, V and $W \subseteq Q$ be finite multisets of transitions. Then we will denote $(M,Q)\,[U+V^+ + W^-\rangle\,(M',Q')$ if, for every $s \in S$,

$$\begin{aligned} M(s) &\geq \sum_{t \in U+V} W(s,t)\cdot(U(t)+V(t)) \\ M'(s) &= M(s) - \sum_{t\in U+V} W(s,t)\cdot(U(t)+V(t)) + \sum_{t \in U+W} W(t,s)\cdot(U(t)+W(t)) \end{aligned}$$

and, furthermore, $Q' = Q+V-W$. The notions of safeness, n-boundedness, cleanness, etc, extend to the case of complex markings. For instance, (M,Q) is safe if for every s

in S,

$$M(s)+\sum_{t\in Q} Q(t)\cdot \max(W(s,t),W(t,s)) \ \leq\ 1.$$

implying (from the T-restrictedness) that Q must be a set of independent transitions, with unitary adjacent arcs, and for all $s\in {}^\bullet Q\cup Q^\bullet$, $M(s)=0$.

The composition operation specified by Ω, provided with a safe complex marking (M,Q) (or, in general, any complex marking such that Q is a set), will be applicable to every Ω-tuple $\vec{\Sigma}$, as defined above, such that Σ_v is static whenever $v\notin Q$, and dynamic otherwise.

4.2.2 Place and transition names of operator and plain boxes

As far as net refinement as such is concerned, the names (identities) of newly constructed transitions and places are irrelevant, provided that we always choose them fresh. However, in our approach to solving recursive definitions on boxes (see [10]), it is the *names* of places and transitions which play a crucial role since we use them to define the inclusion order on the domain of labelled nets. A key to our construction of recursive nets is the use of labelled trees as place and transition names.[9]

First of all, we shall assume that there are two disjoint infinite sets of place and transition names, $\mathsf{P_{root}}$ and $\mathsf{T_{root}}$. Each name $\eta\in\mathsf{P_{root}}\cup\mathsf{T_{root}}$ (or a pair (t,α), where t is a name in $\mathsf{T_{root}}$ and α is a label in Lab) can be viewed as a special tree with a single root labelled with η (or (t,α)) which is also a leaf. Moreover, we shall allow trees as transition and place names, and use a linear notation to express those trees. To this end, the expression $x\lhd\mathcal{S}$, where x is a single root tree labelled with x (x is a name η or a pair (t,α)) and $\mathcal{S}$ is a multiset of trees, is a new tree where the trees of the multiset are appended (with their multiplicity) to the root. Moreover, if $\mathcal{S}=\{p\}$ is a singleton multiset then $x\lhd\mathcal{S}$ will simply be denoted by $x\lhd p$, and if $\mathcal{S}$ is the empty multiset then $x\lhd\mathcal{S}=x$.

We shall further assume that in every operator box, all places and transitions are simply names (i.e., single root trees) from respectively $\mathsf{P_{root}}$ and $\mathsf{T_{root}}$. For the plain boxes, the trees used as names may be more complex. Each transition tree is a finite tree labelled with elements of $\mathsf{T_{root}}$ (at the leaves) and $\mathsf{T_{root}}\times\mathsf{Lab}$ (elsewhere), and each place tree is a possibly infinite (in depth) tree labelled with names from $\mathsf{P_{root}}$ and $\mathsf{T_{root}}$, which has the form:

$$t_1\lhd t_2\lhd\ldots\lhd t_n\lhd s\lhd\mathcal{S},$$

where $t_1,\ldots,t_n\in\mathsf{T_{root}}$ ($n\geq 0$) are transition names and $s\in\mathsf{P_{root}}$ is a place name (so that no confusion will be possible between transition-trees and place-trees: the latter always have a label from $\mathsf{P_{root}}$ and the former never). We comprise all these trees (including the basic ones consisting only of a root as special cases) in our sets of allowed transition and place names, denoted respectively by $\mathsf{T_{tree}}$ and $\mathsf{P_{tree}}$. The definition of net refinement will be done in such a way that, provided all names occurring in Ω are single root trees, and all names occurring in $\vec{\Sigma}$ are in the sets of allowed names, then all names in $\Omega(\vec{\Sigma})$ belong there, too.

[9]We define such labelled trees up to isomorphism.

4.2.3 Formal definition of net refinement $\Omega(\vec{\Sigma})$

Under the assumptions made earlier about the operator box Ω and the Ω-tuple of plain boxes $\vec{\Sigma}$, the result of a simultaneous substitution of the boxes $\vec{\Sigma}$ for the transitions in Ω is a labelled net $\Omega(\vec{\Sigma})$ whose components are defined below.

The set of places of $\Omega(\vec{\Sigma})$ is defined as the (disjoint) union

$$\begin{aligned} S_{\Omega(\vec{\Sigma})} &= \bigcup_{v\in T_\Omega} \mathsf{ST}^v_{\mathsf{new}} \quad \cup \quad \bigcup_{s\in S_\Omega} \mathsf{SP}^s_{\mathsf{new}} \\ &= \bigcup_{v\in T_\Omega} \{\, v\lhd i \mid i\in\ddot{\Sigma}_v\} \quad \cup \quad \bigcup_{s\in S_\Omega} \{\, s\lhd(\{v\lhd x_v\}_{v\in{}^\bullet s} + \{w\lhd e_w\}_{w\in s^\bullet}) \mid \\ &\qquad\qquad x_v \in \Sigma_v^\circ \wedge e_w \in {}^\circ\Sigma_w \,\}. \end{aligned}$$

If s is an isolated place, i.e., ${}^\bullet s = s^\bullet = \emptyset$, then, by the definition of the tree appending operation, $\mathsf{SP}^s_{\mathsf{new}} = \{s\}$. Notice that the multiset of trees in the definition of a place

$$p \;=\; s\lhd(\{v\lhd x_v\}_{v\in{}^\bullet s} + \{w\lhd e_w\}_{w\in s^\bullet}) \;\in\; \mathsf{SP}^s_{\mathsf{new}} \tag{15}$$

is in fact a set since in case there is a side-loop between s and $v = w$ then $x_v \neq e_w$ because x_v is an exit place and e_w is an entry place of Σ_v.

The marking of a place p in $\Omega(\vec{\Sigma})$ is defined in the following way:

$$M_{\Omega(\vec{\Sigma})}(p) = \begin{cases} M_v(i) & \text{if } p = v\lhd i \in \mathsf{ST}^v_{\mathsf{new}} \\ M_\Omega(s) + \sum\limits_{v\in{}^\bullet s} M_v(x_v) + \sum\limits_{w\in s^\bullet} M_w(e_w) & \text{if } p \in \mathsf{SP}^s_{\mathsf{new}} \text{ is as in (15).} \end{cases} \tag{16}$$

Note that if $s \in S_\Omega$ is an isolated place then $M_{\Omega(\vec{\Sigma})}(s) = M_\Omega(s)$.

The set of transitions of $\Omega(\vec{\Sigma})$ is defined as the union

$$T_{\Omega(\vec{\Sigma})} = \bigcup_{v\in T_\Omega} \mathsf{T}^v_{\mathsf{new}} = \bigcup_{v\in T_\Omega} \{(v,\alpha)\lhd R \mid R \in \mathsf{mult}(T_v) \wedge (\lambda_v(R),\alpha) \in \lambda_\Omega(v)\}. \tag{17}$$

The multiset R in $(v,\alpha)\lhd R$ will never be empty since no pair in $\lambda_{\Sigma_v}(v)$ has the empty multiset as its left argument.

The label of a place or transitions x in $\Omega(\vec{\Sigma})$ is defined in the following way:

$$\lambda_{\Omega(\vec{\Sigma})}(x) \;=\; \begin{cases} \mathsf{i} & \text{if } x \in \mathsf{ST}^v_{\mathsf{new}} \\ \lambda_\Omega(s) & \text{if } x \in \mathsf{SP}^s_{\mathsf{new}} \\ \alpha & \text{if } x = (v,\alpha)\lhd R \in \mathsf{T}^v_{\mathsf{new}}. \end{cases}$$

For a place p and transition $u = (v,\alpha)\lhd R$ in $\Omega(\vec{\Sigma})$, the weight function is given by:

$$W_{\Omega(\vec{\Sigma})}(p,u) = \begin{cases} \sum\limits_{t\in R} W_v(i,t)\cdot R(t) & \text{if } p = v\lhd i \in \mathsf{ST}^v_{\mathsf{new}} \\ \sum\limits_{t\in R} W_v(e_v,t)\cdot R(t) & \text{if } p \in \mathsf{SP}^s_{\mathsf{new}} \text{ is as in (15) and } v \in s^\bullet \\ 0 & \text{otherwise,} \end{cases} \tag{18}$$

$$W_{\Omega(\vec{\Sigma})}(u,p) = \begin{cases} \sum\limits_{t\in R} W_v(t,i)\cdot R(t) & \text{if } p = v\lhd i \in \mathsf{ST}^v_{\mathsf{new}} \\ \sum\limits_{t\in R} W_v(t,x_v)\cdot R(t) & \text{if } p \in \mathsf{SP}^s_{\mathsf{new}} \text{ is as in (15) and } v \in {}^\bullet s \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

For p belonging to $\mathsf{SP}^s_{\mathsf{new}}$, we have taken into account the fact that in a box (in this case, in Σ_v) the entry places have no incoming arcs and exit places have no outgoing arcs; otherwise, we would have also introduced terms of the form $W_{\Sigma_v}(x_v,t)\cdot R(t)$ in

$W_{\Omega(\vec{\Sigma})}(p,u)$, and $W_{\Sigma_v}(t,e_v)\cdot R(t)$ in $W_{\Omega(\vec{\Sigma})}(u,p)$). Note that the trees created as place and transition names of $\Omega(\vec{\Sigma})$ obey the constraints formulated in section 4.2.2. Figure 15 shows an example of net refinement, together with all the newly created place and transition (tree) names. Using the linear notation to express tree names we have, for example, $p_1 = 1\lhd\{v_1\lhd 4, v_2\lhd 8\}$, $p_2 = v_1\lhd 6$, $p_3 = 2\lhd v_1\lhd 7$, $t_1 = (v_1,\delta)\lhd\{t,u\}$ and $t_2 = (v_1,\beta)\lhd u$.

4.2.4 Some remarks on net refinement

One might ask why we allow multisets of transitions to be combined, and not only sets, since all combinations of true multisets will lead to arc weights greater than 1, and hence, in the safe context, to dead transitions which could be dropped. The answer is twofold.

First, our aim is to develop the model in such a way that it will not need a major redesign if we wanted to extend it, say, by allowing other initial markings (for instance, a net corresponding to a generalised dynamic expression $\overline{\overline{E}}$ could have two tokens in each of the entry places). Then our argument about some of the transitions being dead would no longer hold, and it seems reasonable to define the refinement in a fully general way, while possibly dropping the dead transitions afterwards, when it is convenient or desirable for other reasons.

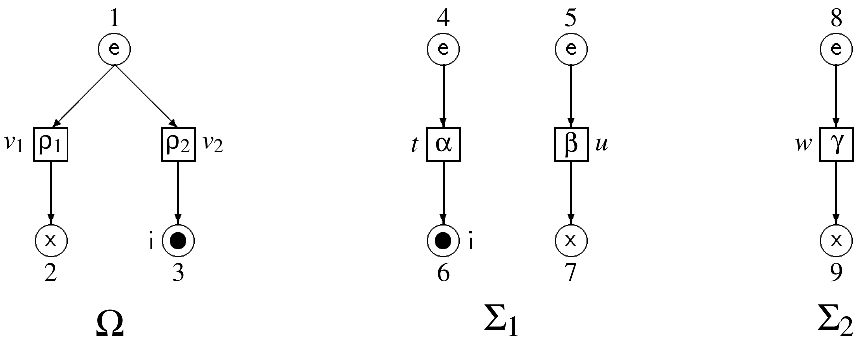
Another, more subtle, argument arises in the context of the structural equivalence iso_T considered in section 4.1.3. With it, the two nets Σ_1 and Σ_2 in figure 16 are equivalent, since they only differ by 'duplicate' transitions and hence will always have the same behaviour, i.e., more precisely, have isomorphic (labelled) box transition systems. However, their synchronisations through the rule using sets rather than multisets (and the standard PBC synchronisation explained in section 3.2.4) would not produce nets which are equivalent with respect to the same kind of equivalence: the synchronisation of Σ_1 with respect to a has the same structure as Σ_1 itself (since no new transition is created), while the synchronisation of Σ_2 gives rise to a new (non-duplicate) transition. This problem does not occur when one adopts the multiset rule. Hence we shall use the multiset based refinement, despite the fact that it may introduce dead transitions in our safe framework.

At this point, it is not certain that, if we start from static boxes, the result of a refinement will be a static box. And indeed, it will turn out that some extra conditions need to be introduced. However, the mechanism we have described is compositional since the definition is based on, and respects, the structure of the components.

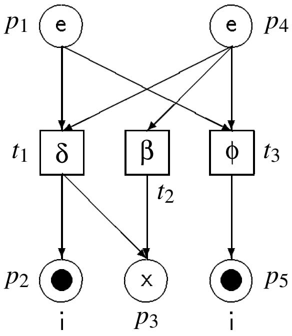
Proposition 4.3 The net $\Omega(\vec{\Sigma})$ defined in section 4.2.3 is a plain box. ■ 4.3

It may also be observed that the result is unmarked if and only if so are Ω and each Σ_v. The T-restrictedness and ex-restrictedness are closely related by the refinement operation. To see it more clearly, let us consider the example shown in figure 17.

While Ω and Σ_2 are well formed, Σ_1 has no exit place and hence is not ex-restricted. The result of the refinement, $\Omega(\vec{\Sigma})$, where $\vec{\Sigma} = (\Sigma_1,\Sigma_2)$, is not T-restricted since the place 2 of Ω does not give rise to any place in the refined net (all the places with a root labelled 2 should have a leaf labelled by an exit place of Σ_1, and there are none). It may also be noticed that, while all the original nets are safe, the result is not even bounded, even if

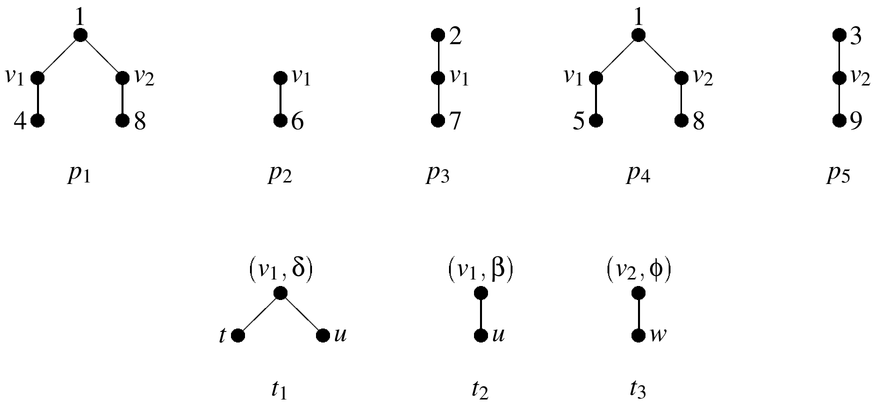


Operator box with $\rho_1 = \{(\{\alpha,\beta\},\delta),(\{\beta\},\beta)\}$, $\rho_2 = \{(\{\gamma\},\phi)\}$, and two plain boxes.



$\Omega(\Sigma_1,\Sigma_2)$

The result of net refinement.



Place and transition trees in the refined net $\Omega(\Sigma_1,\Sigma_2)$.

Figure 15: An example of net refinement.

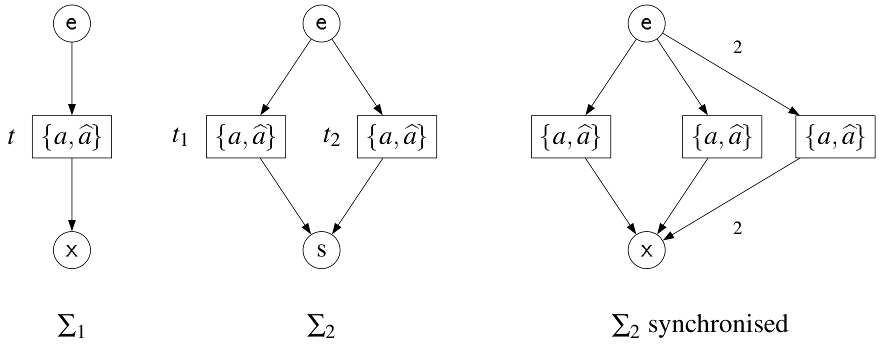


Figure 16: Problem with sets w.r.t. multisets for grouping transitions.

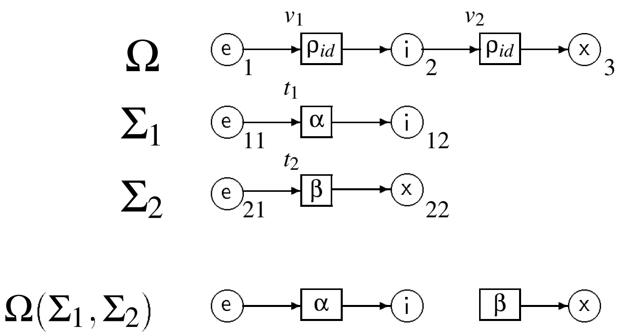


Figure 17: The connection between T-restrictedness and ex-restrictedness.

we start from the empty marking. This shows the importance of the T-restrictedness and ex-restrictedness properties in order to maintain a desirable behaviour of the constructed nets.
It may also be asked why, while we tried to be as general as possible in defining the refinement mechanism, we decided to only consider simple nets for the operator box Ω. This is due to the fact that since, at the end, we shall develop theory only for safe nets, operator boxes with arc weights greater than 1 are not really interesting – while the mechanism may indeed be extended to the non-simple case. A fuller discussion of this point can be found in [8].
Finally, we remark that the notion of refinement defined above works even if the box Ω has side-conditions, and that net isomorphism is preserved through net refinement.

4.3 The Petri net semantics of PBC

We now apply the general net refinement operation to the translation of the PBC expressions presented in sections 2 and 3 into boxes. The translation, represented formally by a mapping $\mathsf{box}_{\mathrm{PBC}}$, will be purely syntactic at this point; later, in section 5, we shall argue that the translation indeed yields static and dynamic boxes, with the same behaviour as that specified by the operational semantics in section 3. The translation will be given explicitly for the static and dynamic basic expressions, and homomorphically (i.e., compositionally) for the remaining ones. To begin with, the translation for the initial and terminal dynamic expressions is given using the translation for the corresponding static expressions, by $\mathsf{box}_{\mathrm{PBC}}(\overline{E}) = \overline{\mathsf{box}_{\mathrm{PBC}}(E)}$ and $\mathsf{box}_{\mathrm{PBC}}(\underline{E}) = \underline{\mathsf{box}_{\mathrm{PBC}}(E)}$, i.e., by simply putting a single token in each entry place, and a single token in each exit place, respectively. Moreover, with each PBC operator OP of arity n we shall associate an operator box Ω_{OP} with n transitions, and define, for the various combinations of static and dynamic PBC expressions:

$$\mathsf{box}_{\mathrm{PBC}}(OP(H_1,\ldots,H_n)) = \Omega_{OP}(\mathsf{box}_{\mathrm{PBC}}(H_1),\ldots,\mathsf{box}_{\mathrm{PBC}}(H_n))$$

where the H_i's are static and dynamic expressions such that $OP(H_1,\ldots,H_n)$ conforms to the syntax given in sections 3.1 and 3.3.4.

4.3.1 The elementary actions

The translation rule for the basic PBC expressions $\alpha \in \mathsf{Lab}_{\mathrm{PBC}}$ is $\mathsf{box}_{\mathrm{PBC}}(\alpha) = \mathsf{base}_\alpha$, where base_α is the static box shown in figure 18.

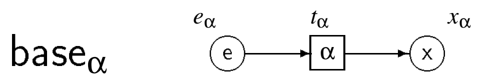


Figure 18: Static box for the basic expression α.

It may be observed that, when using the labelled version of step sequence semantics, we have $\mathsf{box}_{\mathrm{PBC}}(\overline{\alpha})\ [\emptyset\rangle_{\mathsf{lab}}\ \mathsf{box}_{\mathrm{PBC}}(\overline{\alpha})$ and $\mathsf{box}_{\mathrm{PBC}}(\overline{\alpha})\ [\{\alpha\}\rangle_{\mathsf{lab}}\ \mathsf{box}_{\mathrm{PBC}}(\underline{\alpha})$, similarly as in the SOS semantics, where $\overline{\alpha} \xrightarrow{\emptyset} \overline{\alpha}$ by IN1 and $\overline{\alpha} \xrightarrow{\{\alpha\}} \underline{\alpha}$ by the action rule. This example gives some flavour of the anticipated consistency result.

4.3.2 Parallel composition, choice, and sequential composition

The operator box associated with the parallel composition of PBC is given in figure 19.

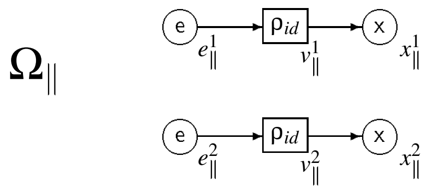


Figure 19: Operator box for parallel composition.

Here as elsewhere, we use descriptive names such as $e^1_{\|}$ for the entry place of the first component of the parallel composition. This operator box will create two separate copies of the operands Σ_1 and Σ_2 which are refined into $v^1_{\|}$ and $v^2_{\|}$, respectively. By employing the tree device used in net refinement, this is allowed even if Σ_1 and Σ_2 are the same net. Note that we have $\mathsf{box}_{\mathrm{PBC}}(\overline{a\|b}) = \mathsf{box}_{\mathrm{PBC}}(\overline{a}\|\overline{b})$ in the sense of labelled net equality, not only net isomorphism, and the inaction rule IPAR1 gives $\overline{a\|b} \xrightarrow{\emptyset} \overline{a}\|\overline{b}$. We shall see later that this is not a coincidence, but an instance of a more general fact: as already explained, the inaction rules should not be interpreted as denoting anything actually happening in the semantic sense, but rather as describing two different views of the same system, and the arrow $\xrightarrow{\emptyset}$ as denoting a change from one point of view to another. Finally, one can show that parallel composition is *commutative*, $(\Sigma\|\Theta)\ \mathsf{iso}\ (\Theta\|\Sigma)$, and *associative*, $(\Sigma\|(\Theta\|\Psi))\ \mathsf{iso}\ ((\Sigma\|\Theta)\|\Psi)$. Note that we write $\Sigma\|\Theta$ instead of $\Omega_{\|}(\Sigma,\Theta)$, etc. This notational convention will be used also in the case of other PBC operators and operator boxes.

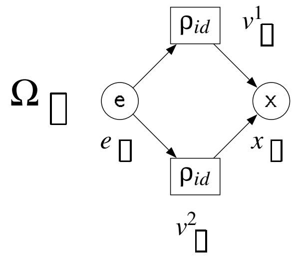
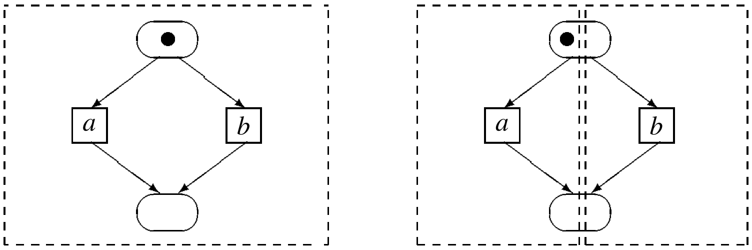


Figure 20: Operator box for choice composition.

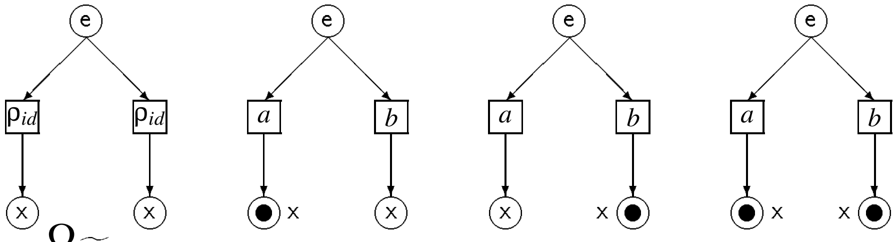
The operator box associated with the choice operator of PBC is given in figure 20. Note that we have $\mathsf{box}_{\mathrm{PBC}}(\overline{a \square b}) = \mathsf{box}_{\mathrm{PBC}}(\overline{a} \square b)$, and that this corresponds to the inaction rule IC1L, $\overline{a \square b} \xrightarrow{\emptyset} \overline{a} \square b$. This is further illustrated in figure 21. The rule says that the initial marking of a choice expression can be interpreted as the initial marking of its left constituent (as well as of its right constituent). In general, choice is commutative, $(\Sigma \square \Theta)\,\mathsf{iso}\,(\Theta \square \Sigma)$, and associative, $(\Sigma \square (\Theta \square \Psi))\,\mathsf{iso}\,((\Sigma \square \Theta) \square \Psi)$.



$\text{box}_{\text{PBC}}(\overline{a\,\square\, b})$ $\text{box}_{\text{PBC}}(\overline{a}\,\square\, b)$

Figure 21: Two different views of the same system (with respect to $\square$).

Notice that the two rules $\underline{E}\,\square\, F \xrightarrow{\emptyset} \underline{E\,\square\, F}$ and $E\,\square\,\underline{F} \xrightarrow{\emptyset} \underline{E\,\square\, F}$ exclude the net $\Omega_{\tilde{\square}}$ shown in figure 22 from being an adequate operator box for the choice composition. The reason is that in $\Omega_{\tilde{\square}}(\mathsf{base}_a, \mathsf{base}_b)$, the marking after an execution of the a-labelled transition cannot be interpreted as the same as the marking after an execution of the b-labelled transition, and that neither is equal to the exit marking. But, by the inaction rules IC2L and IC2R, it is required that the branches of a choice construct re-join after the initial branching.



$\Omega_{\tilde{\square}}$ transl. of $\underline{a}\,\square\, b$ transl. of $a\,\square\,\underline{b}$ transl. of $\underline{a\,\square\, b}$

Figure 22: $\Omega_{\tilde{\square}}$ is not a good operator box for choice composition.

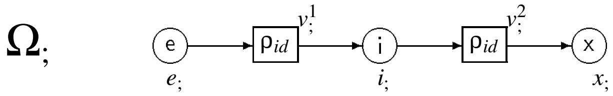


Figure 23: Operator box for sequential composition.

The operator box associated with the sequential operator of PBC is shown in figure 23. It is perhaps instructive to examine the Petri net corresponding to $\overline{(a;b)}\,\square\, a$, shown in figure 24. By the operational semantics, this expression can execute the step sequence $\{\{a\}\}\{\{b\}\}$. The same sequence is also possible in the Petri net, by executing the upper a-labelled transition first, and then the b-labelled transition. As before, the actual changes of marking take place only in the non-$\xrightarrow{\emptyset}$ derivation steps, and all the $\xrightarrow{\emptyset}$ derivations correspond to different interpretations of the same system. Sequential composition is associative, $(\Sigma; (\Theta; \Psi))$ iso $((\Sigma; \Theta); \Psi)$.

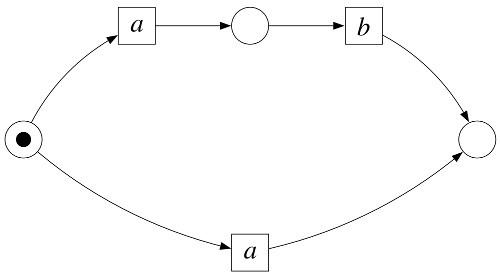


Figure 24: The box corresponding to $\overline{(a;b) \square a}$.

4.3.3 Synchronisation

The different variants of the SOS semantics for the synchronisation of the PBC expressions (see section 3.2.4) are captured by various relabellings. We will only describe the standard PBC synchronisation in detail. Its operator box has a two-place, one-transition structure (see figure 25), where $\rho_{\mathsf{sy}\,a}$ is the smallest relabelling which contains ρ_{id} and such that, if $(\Gamma, \alpha+\{a\}) \in \rho_{\mathsf{sy}\,a}$ and $(\Delta, \beta+\{\widehat{a}\}) \in \rho_{\mathsf{sy}\,a}$, then $(\Gamma+\Delta, \alpha+\beta) \in \rho_{\mathsf{sy}\,a}$, as well.

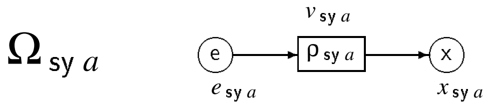


Figure 25: Operator box for standard synchronisation.

The net semantics of standard synchronisation may be compared with the operational semantics of the same operator, where all the semantic effect is contained in the derivation rules, while the $\equiv$-equivalences (i.e., $\emptyset$-derivations) are of the most basic format. This observation is general. Control flow operators (such as parallel composition and sequential composition) tend to have simple context rules and more complex $\equiv$-rules, and also simple transition relabellings (namely, the identity relabelling) but more complex operator boxes. On the other hand, communication interface operators (such as synchronisation, basic relabelling and restriction) tend to have more complex context rules, simple $\equiv$-rules, not so simple transition relabellings, but very simple operator boxes.

We illustrate this synchronisation operator in figure 26, giving the translation for the expression $(a \| \widehat{a})$ sy a. The left-hand and right-hand transitions are present because $(\{\{a\}\}, \{a\})$ and $(\{\{\widehat{a}\}\}, \{\widehat{a}\})$ belong to $\rho_{id} \subseteq \rho_{\mathsf{sy}\,a}$, while the middle transition is there due to the last clause in the definition of $\rho_{\mathsf{sy}\,a}$ given above. The translation is correct with respect to the SOS semantics, because only the pairs $(\{\{a\}\}, \{a\})$, $(\{\{\widehat{a}\}\}, \{\widehat{a}\})$ and $(\{\{a\}, \{\widehat{a}\}\}, \emptyset)$ in $\rho_{\mathsf{sy}\,a}$ are relevant, as only they can be matched by the labels of some multiset of transitions of $\mathsf{box}_{\mathrm{PBC}}(a \| \widehat{a})$.

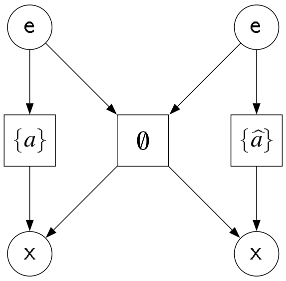


Figure 26: The box of $(\{a\}\|\{\widehat{a}\})$ sy a.

An alternative characterisation of $\rho_{\mathsf{sy}\,a}$ is given by the following formula:

$$\begin{aligned}\rho_{\mathsf{sy}\,a} \;=\; & \rho_{id} \cup \{(\sum_{i=1}^{n}\{\alpha_i\}, \sum_{i=1}^{n}\alpha_i-(n-1)\cdot\{a,\widehat{a}\}) \mid \\ & n\geq 2 \wedge \prod_{i=1}^{n}(\alpha_i(a)+\alpha_i(\widehat{a}))>0 \wedge \sum_{i=1}^{n}\alpha_i(a)\geq n-1 \wedge \sum_{i=1}^{n}\alpha_i(\widehat{a})\geq n-1\}.\end{aligned}$$

Moreover, $((\Sigma\,\mathsf{sy}\,a)\,\mathsf{sy}\,b)\ \mathsf{iso}\ ((\Sigma\,\mathsf{sy}\,b)\,\mathsf{sy}\,a)$ and $((\Sigma\,\mathsf{sy}\,a)\,\mathsf{sy}\,a)\ \mathsf{iso}_T\ (\Sigma\,\mathsf{sy}\,a)\ \mathsf{iso}\ (\Sigma\,\mathsf{sy}\,\widehat{a})$. Idempotence needs iso_T instead of iso, since a double application of the synchronisation operation will add twice the same (i.e., duplicating) groups of synchronised transitions. The relabelling $\rho_{\mathsf{sy}\,a}$, while corresponding to the smallest binary multiway extension of CCS synchronisation (as discussed in section 2.5), exhibits at the same time some problems. Even if we start from a finite net, the result of the synchronisation may be infinite; for instance, in the case of the net modelling the expression $\{a,\widehat{a},b\}$ sy a. The next result provides a characterisation of all such cases.

Theorem 4.4 Let Σ be a box with finitely many transitions. Then Σ sy a has infinitely many transitions if and only if one of the following holds:

- There are distinct transitions u and w in Σ such that $\lambda(u)(a)>1$ and $\lambda(w)(\widehat{a})>1$.
- There is a transition v in Σ such that $\lambda(v)(a)>0$ and $\lambda(v)(\widehat{a})>0$. □

To model, in addition, the rule SY3, we simply change the relabelling of the operator box $\Omega_{\mathsf{sy}\,a}$ from $\rho_{\mathsf{sy}\,a}$ to $\rho_{\mathsf{sy}'a}$, where $\rho_{\mathsf{sy}'a}$ is the smallest relabelling containing $\rho_{\mathsf{sy}\,a}$ and such that, if $(\Gamma,\alpha+\{a,\widehat{a}\})\in\rho_{\mathsf{sy}'a}$ then also $(\Gamma,\alpha)\in\rho_{\mathsf{sy}'a}$. The resulting synchronisation satisfies similar laws as the previous one. The treatment of the step-synchronisation, as described by the additional SOS rule SY4, is more delicate; the interested reader is referred to [8].

4.3.4 Basic relabelling, restriction and scoping

For the basic relabelling, all the semantic effect of the operator box (see figure 27) is contained in the relabelling $\rho_{[f]}$ annotating the only transition, $v_{[f]}$. Assuming that f is a conjugate-preserving function $f\colon \mathsf{A}_{\mathrm{PBC}}\to\mathsf{A}_{\mathrm{PBC}}$, extended to $\mathsf{Lab}=\mathsf{Lab}_{\mathrm{PBC}}$, $\rho_{[f]}$ is the relation containing all pairs of the form $(\{\alpha\},f(\alpha))$, for $\alpha\in\mathsf{Lab}_{\mathrm{PBC}}$. One can show that $(\Sigma[f])[g]\ \mathsf{iso}_T\ \Sigma[g\circ f]$ and $\Sigma[\rho_{id}]\ \mathsf{iso}\ \Sigma$.

The net semantics of a restricted expression is almost self-explanatory; the net of E rs a is simply the net of E where all transitions whose labels carry at least one a or at least

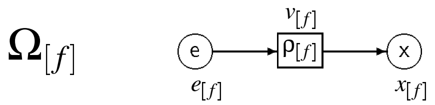


Figure 27: Operator box for basic relabelling.

one $\widehat{a}$ are erased. This corresponds to the operator box shown in figure 28, where $\rho_{\mathsf{rs}\,a} = \rho_{\mathsf{mult}(\mathsf{A}_{\mathrm{PBC}}\setminus\{a,\widehat{a}\})}$ is the restriction on the (multi-)labels without a or $\widehat{a}$.

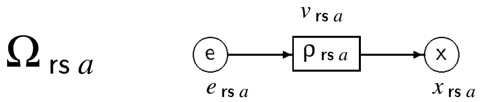


Figure 28: Operator box for restriction.

Notice that this may create boxes without any transition. Boxes whose transition sets and sets of internal places are empty will be called *stop-boxes* (see figure 29).

Figure 29: The simplest stop-box.

We have that $((\Sigma\ \mathsf{rs}\ a)\ \mathsf{rs}\ b)\ \mathsf{iso}\ ((\Sigma\ \mathsf{rs}\ b)\ \mathsf{rs}\ a)$ and $((\Sigma\ \mathsf{rs}\ a)\ \mathsf{rs}\ a)\ \mathsf{iso}\ (\Sigma\ \mathsf{rs}\ a)\ \mathsf{iso}\ (\Sigma\ \mathsf{rs}\ \widehat{a})$.
By combining the two previous operators, we obtain the operator box for scoping shown in figure 30, where $\rho_{[a:]} = \rho_{\mathsf{sy}\,a} \cap \{(\Gamma,\alpha) \mid \alpha(a) = 0 = \alpha(\widehat{a})\}$.
Then $[a:[b:\Sigma]]\ \mathsf{iso}\ [b:[a:\Sigma]]$ as well as $[a:[a:\Sigma]]\ \mathsf{iso}_T\ [a:\Sigma]\ \mathsf{iso}\ [\widehat{a}:\Sigma]$.

4.3.5 Iteration

The treatment of the iteration operator $[E*F*E']$ deserves special attention. One possibility would be to associate with it the operator box $\Omega^2_{[**]}$ shown in figure 31.

Here, $v^{2,1}_{[**]}$ corresponds to E, $v^{2,2}_{[**]}$ corresponds to F, and $v^{2,3}_{[**]}$ corresponds to E'. However, this operator does not preserve the safeness of the static and dynamic boxes, as it can be demonstrated by taking the PBC expression $[a*(b\|a)*a]$, whose translation is shown in figure 32. It may be observed that, if we start from the entry marking, after the execution of t_1 and t_2, the place s_{21} will receive two tokens. The net is thus 2-bounded, but not safe, as desired. On the level of executions, the sequential, as well as the step, behaviour of the net is as expected but if we look more closely at the partial order semantics it occurs that unwanted dependencies may appear. Thus the operator $\Omega^2_{[**]}$ is not fully satisfactory in general, and we should search for another one.
A careful analysis of the reasons of this phenomenon (see [15, 16, 8]) shows that the situation is never worse than that exhibited by the example in figure 32, i.e., the boxes obtained for the PBC expressions with the operator $\Omega^2_{[**]}$ are always 2-bounded. Thus, for instance, $[a*(c\|b\|a)*a]$ is 2-bounded as well, even though the middle part is a 3-way parallel composition. Moreover, the 2-boundedness is 'free of auto-concurrency', in the sense that each transition always has at least one safe pre-place and at least one safe post-place. And, finally, the origin of non-safeness can be traced to the fact that the

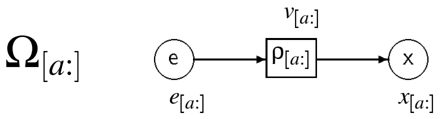


Figure 30: Operator box for scoping.

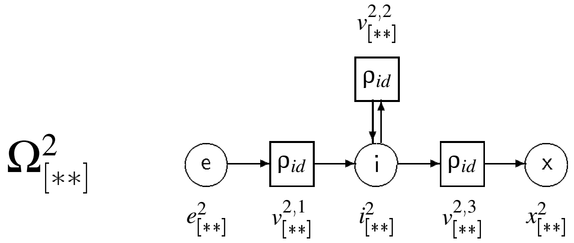


Figure 31: Operator box for iteration (2-bounded version).

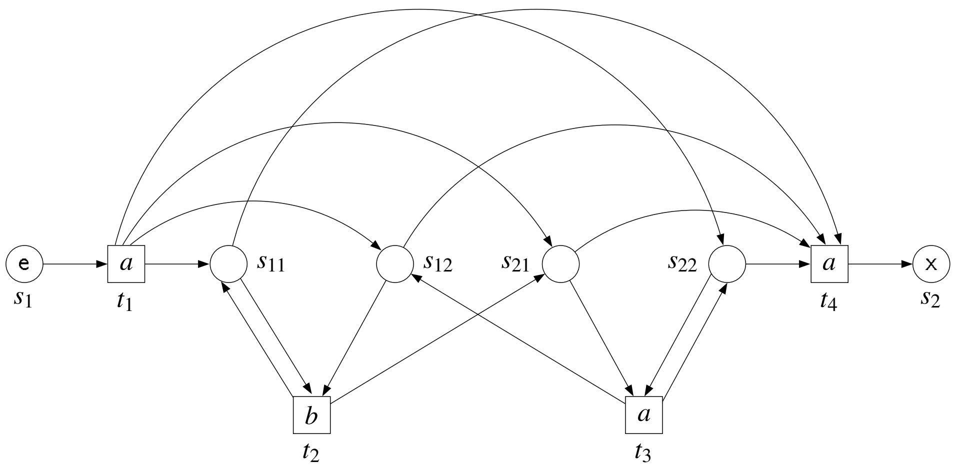


Figure 32: The net $\mathsf{box}^2_{\mathrm{PBC}}([a*(b\|a)*a])$.

operator $\Omega^2_{[**]}$ has a side-loop (involving $i^2_{[**]}$ and $v^{2,2}_{[**]}$) and that the operand replacing $v^{2,2}_{[**]}$ has disjoint, independently starting and terminating subnets, like that modelling $b\|a$. If we do not want to abandon the modelling of general parallel composition, the only way to guarantee safeness is to get rid of the side-condition. The idea we shall exploit is to perform an unfolding of the loop of the operator $\Omega^2_{[**]}$ which leads to the operator box $\Omega_{[**]}$ shown in figure 33.

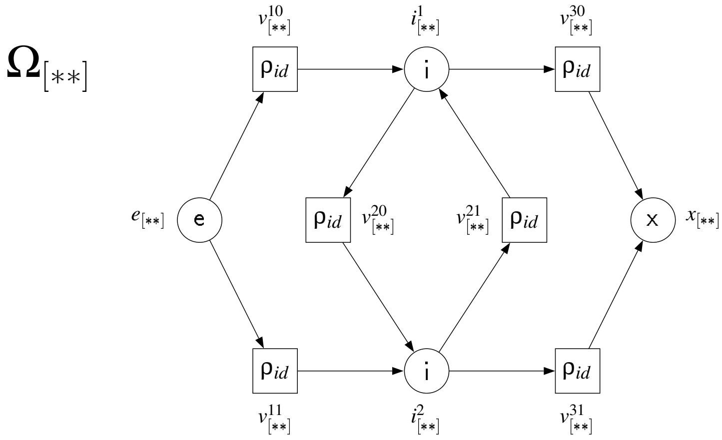


Figure 33: Operator box for iteration (the safe version).

The translation for the static PBC expression $[E*F*E']$ will be given by

$\mathsf{box}_{\mathrm{PBC}}([E*F*E']) =$
$\Omega_{[**]}(\mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(F), \mathsf{box}_{\mathrm{PBC}}(E'), \mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(F), \mathsf{box}_{\mathrm{PBC}}(E')),$

where the ordering of the transitions of $\Omega_{[**]}$ is $v^{10}_{[**]}, v^{20}_{[**]}, v^{30}_{[**]}, v^{11}_{[**]}, v^{21}_{[**]}, v^{31}_{[**]}$. It may be noticed that one of the transitions $v^{10}_{[**]}$ and $v^{11}_{[**]}$ is not strictly necessary for obtaining the desired behaviour, but we keep both just to maintain the symmetry of the operator box. Strictly speaking, the operator $\Omega_{[**]}$ has the arity of 6, but we turned it into a ternary one by restricting the allowable combinations of its operands. The situation is slightly more complex for the translation of the dynamic iterative expressions. There are two different possible translations for $[E*G*E']$, namely, with $F=\lfloor G\rfloor$:

$$\Omega_{[**]}(\mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(G), \mathsf{box}_{\mathrm{PBC}}(E'), \mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(F), \mathsf{box}_{\mathrm{PBC}}(E')),$$
$$\Omega_{[**]}(\mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(F), \mathsf{box}_{\mathrm{PBC}}(E'), \mathsf{box}_{\mathrm{PBC}}(E), \mathsf{box}_{\mathrm{PBC}}(G), \mathsf{box}_{\mathrm{PBC}}(E')).$$

They are $\cong$-equivalent (by a symmetry argument), but not exactly the same. However, this is more acceptable than nonsafeness, and we shall disregard this (small) ambiguity choosing arbitrarily between the two possibilities.

4.3.6 Data expressions

Since the loop-like data expressions are considered as basic processes in PBC, we shall need to provide for them an explicit translation. The translation relies on the value domain D_z of a program variable z, and we first give a simple example of translating a binary variable b in figure 34.

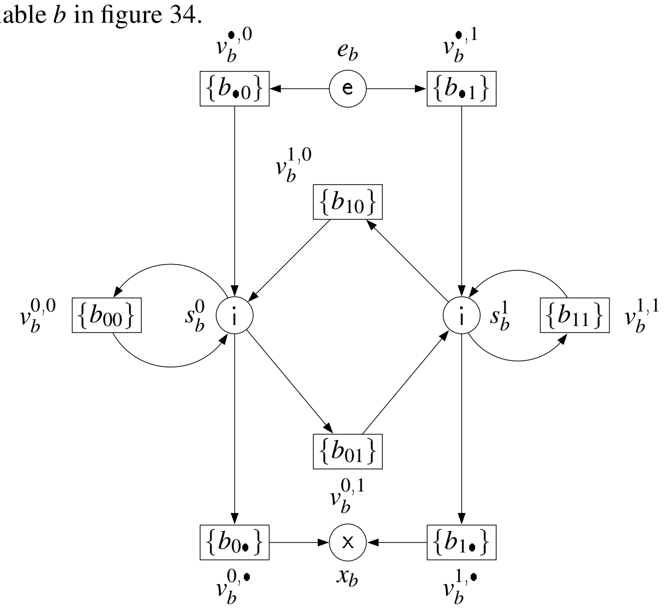


Figure 34: Static box representing a binary variable b.

The general translation proceeds thus. Given a program variable z with the data domain D_z, we define $\mathsf{box}_{\mathrm{PBC}}([z_{()}]) = (S_z, T_z, W_z, \lambda_z, \emptyset)$ where the places and transitions are $S_z = \{e_z, x_z\} \cup \{s_z^u \mid u \in D_z\}$ and $T_z = \{v_z^{\bullet,u}, v_z^{u,\bullet} \mid u \in D_z\} \cup \{v_z^{u,v} \mid u, v \in D_z\}$, the weight function returns 1 for the pairs in the set

$$\{\ (e_z, v_z^{\bullet,u})\ ,\ (v_z^{\bullet,u}, s_z^u)\ ,\ (s_z^u, v_z^{u,v}), (v_z^{u,v}, s_z^v)\ ,\ (s_z^u, v_z^{u,\bullet})\ ,\ (v_z^{u,\bullet}, x_z) \mid u, v \in D_z\}$$

and 0 otherwise; the label function is defined, for all values $u, v \in D_z$, by:

$$\begin{array}{lclclclclcl} \lambda_z(e_z) & = & \mathsf{e} & \quad & \lambda_z(x_z) & = & \mathsf{x} & \quad & \lambda_z(s_z^u) & = & \mathsf{i} \\ \lambda_z(v_z^{\bullet,u}) & = & \{z_{\bullet u}\} & & \lambda_z(v_z^{u,\bullet}) & = & \{z_{u\bullet}\} & & \lambda_z(v_z^{u,v}) & = & \{z_{uv}\}. \end{array}$$

The translation for $[\overline{z_{(u)}}]$ is $\mathsf{box}_{\mathrm{PBC}}([z_{()}])$ with a single token in the place s_z^u. Note that if D_z is infinite, then so is its data box.

4.3.7 Generalised operators

The operator boxes for the generalised control flow operators (section 3.3.3) are very straightforward if the index sets are finite, but less so otherwise. We omit the discussion at this point; the reader is referred to [8].

4.3.8 Generalised iterations

In this section we turn to the (binary and unary) generalisations of the ternary iteration operator discussed in section 3.3.1. Let us first examine how the problems identified previously are rendered in the Petri net framework. A possible translation for the operator $[E * F\rangle$ would be to use a net operator such as that shown in figure 35.

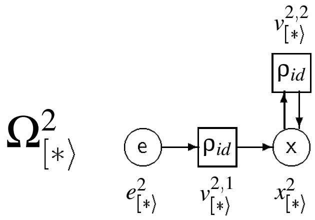


Figure 35: Operator box for binary generalised iteration (2-bounded version).

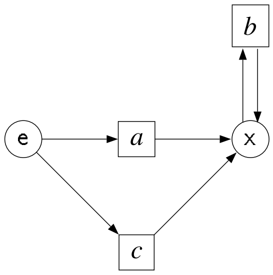


Figure 36: The net $\mathsf{box}^2_{\mathrm{PBC}}([a * b\rangle \, \square \, a)$.

Besides the potential problems due to the side loop which have already been mentioned, this net does not satisfy the ex-directedness constraint required of all the boxes, since there is an arc leaving the (unique) exit place. However, one could ask at this point if this constraint is really necessary; so let us have a further look at the reasons why it has been imposed.

The definition given for the net refinement operation can easily be adapted to work perfectly well with non-ex-directed operators and operands. But if we consider the behaviour we intuitively expect from the resulting nets, there is a problem. Let us consider, for instance, the box we would obtain for $([a * b\rangle \, \square \, c)$, as illustrated in figure 36.

It may be noticed that, from the entry marking, an evolution $\{\{c\}\}\{\{b\}\}$ is allowed, which corresponds to the SOS semantics already considered for this operator, but not to what may be expected from a choice, since the loop is entered 'from the end' after having chosen the other branch of the alternative. The ex-directedness has been explicitly introduced in order to avoid this behaviour in the case of the choice operator. However, if it is ascertained that a loop does not occur terminally in an enclosing choice, or in an enclosing loop, then this particular problem also disappears, and ex-directedness is no longer required.

If we try to get rid of the side condition using the same kind of unfolding as that used to go from $\Omega^2_{[**]}$ to $\Omega_{[**]}$ directly, not only does the problem of non-ex-directedness persist, but also another one emerges, as shown in figure 37. In this net, the exit marking (with a token in each of the exit places) may never be reached, which certainly does not correspond to the expectations. Thus one of the exit places should be turned into an internal one. Since the graph is symmetrical, let us consider the case where $x^1_{[*\rangle}$ becomes internal. Then the symmetry is broken and, if we first perform $v^1_{[*\rangle}$, afterwards

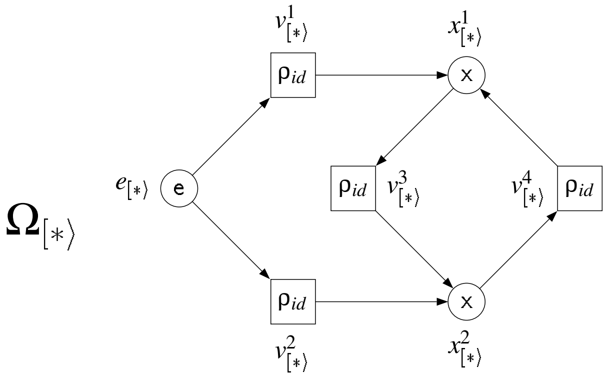


Figure 37: Operator box for binary generalised iteration (safe version).

it will be necessary to perform an odd number of executions of the looping part in order to be able to reach the exit marking (an even number would be necessary if we first chose $v^2_{[*\rangle}$). This is hardly acceptable either since we do not know in advance the parity of the required executions of the looping part.

The reader will easily verify that similar (but not identical) problems occur for the direct translation of the $\langle *]$-operator.

A common solution to these problems is to 'unwind the loop once'. In our context, this solution has the following form. Unwinding $[E * F\rangle$ once means that the case of executing 'E and then zero times F' is separated from the case of executing 'E and then at least once F', i.e.,

$$[E * F\rangle \quad \stackrel{\mathrm{df}}{=} \quad (E \,\square\, [E * F * F]).$$

Similarly, unwinding $\langle F * E']$ once means that the case of executing 'zero times F and then E'' is separated from the case of executing 'at least once F and then E'', i.e.,

$$\langle F * E'] \quad \stackrel{\mathrm{df}}{=} \quad (E' \,\square\, [F * E' * E']).$$

Note, however, that, e.g., $[a * b\rangle$ is not equivalent to $(a \,\square\, [a * b * b])$ under any of the congruences we have defined (beware of the apparently innocent-looking redo/skip transitions), while the two expressions are step equivalent. As we have already seen, this is indicative of the fact that in sufficiently ill-behaved environments, they cannot be exchanged for one another.

The last version of iteration, i.e., the unary $\langle\rangle$-operator, leads to an additional problem. Its most immediate translation would lead to the operator box shown in figure 38. For this box, the labelling is not legal, since it uses a place which should be both an entry and an exit one. While such a generalisation works in some examples, we have chosen not to pursue it in this paper, because many of the results depend strongly on the fact that a place cannot be both an entry place and an exit place. The loop $\langle F\rangle$ is often modelled by adding a silent action where it creates no behavioural problem, e.g., by the loop $\langle F * \emptyset]$ (but neither by $[\emptyset * F\rangle$ nor by $[\emptyset * F * \emptyset]$).

Summarising, the above discussion reassures us that the decision not to incorporate the generalised iteration operators into the standard theory was a justified one, unless their usage is constrained so as not to cause the kind of problems we have discussed.

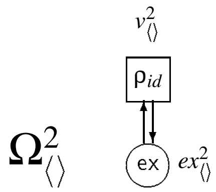


Figure 38: Operator box for unary generalised iteration (2-bounded version).

5 The Box Algebra

We still have to discuss the issue of consistency between the operational and denotational semantics developed for PBC in the previous sections. Instead of pursuing this specific goal, we shall develop a general framework to define process algebras with two consistent semantics, of which PBC and other process algebras will be special cases.

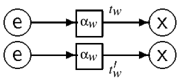
5.1 SOS–operator boxes

Our aim here is to find a class of operator boxes that could be applied to static and dynamic boxes — the base plain boxes considered in the preceding sections — in such a way that SOS rules could also be formulated in the domain of nets. We will define a class of operator boxes satisfying this property, called *sos-operator boxes*, by imposing three conditions on a generic operator box $\Omega = (S, T, W, \lambda, \emptyset)$ which we take to be a candidate sos-operator box.

To simplify the argument, but without losing its generality, we assume that only the identity relabelling ρ_{id} is used to annotate the transitions in Ω. Another point arises from the observation that here a concrete net operator Ω is aimed at modelling a corresponding syntactic operator op_Ω, much in the same way as the net operator $\Omega_;$ was used to model sequential composition. But, whereas Ω might in principle be marked, we have no means to specify explicitly the marking of a symbol representing a net operator, as it is a mere lexical construct or sign. However, from the static or dynamic nature of the various operands, it is possible to associate a marking to an expression, but it will be a purely imaginary one. Hence Ω and, indeed, all the operator boxes considered in this section will have purely imaginary markings which will simply be left implicit.

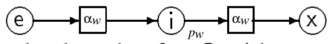
The three defining conditions for Ω to be an sos-operator box, (C1)–(C3) below, will all be derived by applying Ω to a suitably chosen tuple of static boxes $\vec{\Theta}$, and then considering some properties that the composition $\Omega(\vec{\Theta})$ ought to satisfy. The first requirement is formulated thus.

Requirement 1: Operations on static boxes should not lead outside the semantic domain of static boxes. To analyse its consequences, consider an Ω-tuple of very simple static boxes $\Theta_v = (\mathrm{e}) \longrightarrow \boxed{\alpha_v}_{t_v} \longrightarrow (\mathrm{x})$ where $v \in T$ and $\alpha_v \neq \alpha_w$ for $v \neq w$. An easy observation is that, due to the simple form of the refining nets, we can think of $\Omega(\vec{\Theta})$ as though it were Ω with each transition v being labelled by α_v. And so the first requirement implies that $\Omega(\vec{\Theta})$, and thus Ω itself, should be a static box, i.e., all the markings reachable from $^\circ\Omega$, the entry marking of Ω, should be safe and clean: *(C1) Ω is a static box.*

Another consequence of the first requirement is that no transition $w \in T$ that can be enabled at a marking reachable from the entry marking $^\circ\Omega$ is allowed to have a side condition. This can be demonstrated using an argument similar to that used in the discussion of the operator $\Omega^2_{[**]}$ in section 4.3.5 to show that such a side condition can lead to a non-safe marking after performing a suitable net refinement. Indeed, let us replace, in the above Ω-tuple $\vec{\Theta}$ of static boxes, Θ_w by $\Theta'_w =$ 

leaving other Θ_v's unchanged. Then, in $\Omega(\vec{\Theta})$, there will be four places of the same kind as places s_{11}, s_{12}, s_{21} and s_{22} in figure 32, such that at some marking reachable from $^\circ(\Omega(\vec{\Theta}))$ it will be possible to execute the transition $w \lhd t_w$ and the resulting marking will have two tokens in one of these places. In the definition of an sos-operator box we shall use a slightly stronger,[10] but simpler (structural) condition: *(C2)* Ω *is pure*.

The second requirement is directly motivated by our wish to obtain SOS rules for compositionally defined nets.

Requirement 2: A net obtained as a result of an evolution of a net composed from other nets, should itself be a composition of nets related to the original ones. We interpret this as saying that no matter how $\overline{\Omega(\vec{\Theta})}$ evolves, the resulting net Σ should be derivable as a composition $\Sigma = \Omega(\vec{\Theta}')$ where $\lfloor \vec{\Theta}' \rfloor = \lfloor \vec{\Theta} \rfloor$. This leads to a perhaps unexpected property of the complex markings reachable from the entry marking of Ω. For suppose that Ω is a safe operator box and (M,Q) is a complex marking reachable from $^\circ\Omega$. Then, from the safeness assumption made about Ω, M and Q are sets and $M \cap ({}^\bullet Q \cup Q^\bullet) = \emptyset$. Consider yet another Ω-tuple of simple static boxes obtained from the previously used $\vec{\Theta}$ by replacing, for every $w \in Q$, the Θ_w by $\Theta''_w =$ 

leaving other Θ_v's unchanged. Notice that $\Omega(\vec{\Theta})$ can now be thought of as Ω with every transition $v \in T \setminus Q$ being labelled by α_v, and every transition $w \in Q$ being split into the 'beginning of w' and 'end of w', both transitions being labelled by α_w and 'joined' by a single place p_w.

It should be clear that $\overline{\Omega(\vec{\Theta})}$ is a safe net which can evolve into the net Σ whose (safe) marking is $M \cup \{p_w \mid w \in Q\}$. From what we have already said, we should be able to represent Σ as $\Omega(\vec{\Theta}')$ where $\lfloor \vec{\Theta}' \rfloor = \lfloor \vec{\Theta} \rfloor$. Then, from the definition of net refinement and safeness of Σ, it follows that for every $v \in T \setminus Q$, Θ'_v is Θ_v or $\overline{\Theta}_v$ or $\underline{\Theta}_v$, and for every $w \in Q$, Θ'_w is Θ''_w with exactly one token inside the place p_w. And, crucially, M is the disjoint union of the pre-sets of transitions v such that $\Theta'_v = \overline{\Theta}_v$ and the post-sets of transitions v such that $\Theta'_v = \underline{\Theta}_v$. In other words, the 'real' part of the complex marking (M,Q) can be factorised onto pre- and post-sets of some transitions in T.

Following the above observation, we shall say that a pair $\mu = (\mu_e, \mu_x)$ of sets of transitions of Ω is a *factorisation* of a set of places $M \subseteq S$ if M is the disjoint union of the sets ${}^\bullet v$, for all $v \in \mu_e$, and the sets $v^\bullet$, for all $v \in \mu_x$. Then a quadruple of sets of transitions of Ω, $\mu = (\mu_e, \mu_d, \mu_x, \mu_s)$ is a *factorisation* of a complex safe marking (M,Q) of Ω if $\mu_d = Q$, $\mu_s = T \setminus (\mu_e \cup \mu_d \cup \mu_x)$, and (μ_e, μ_x) is a factorisation of M. Ω itself will be called *factorisable* if for every safe complex marking (M,Q) reachable[11] from the entry marking of Ω, there is at least one factorisation. And the third, and final, condition we

[10]If we required, as it would seem natural, that no transition of Ω is dead from $^\circ\Omega$, this would no longer be a strengthening. However, we will refrain from introducing this additional condition for reasons that will be explained later, when we discuss the example in figure 39.

[11]This includes the exit marking $(\Omega^\circ, \emptyset)$ which is considered reachable through the skip transition.

will need is that: *(C3)* Ω *is factorisable*.
To summarise, in this section, we shall only consider operators fulfilling the conditions (C1), (C2) and (C3), i.e., simple pure static (hence safe and clean) factorisable operator boxes, called henceforth *sos-operator boxes*.

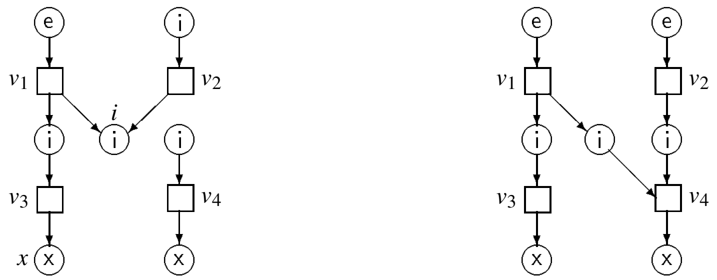


Figure 39: An sos-operator box with a necessary dead transition; and a non-factorisable (but finite, pure and static) box. All transitions are labelled by ρ_{id}.

A factorisation of a marking (M,Q) is essentially a way of representing its real part, M, as the disjoint union of the pre-sets of a set of transitions, μ_e, and the post-sets of another set of transitions, μ_x. Intuitively, μ_e are transitions which can be executed at (M,Q), and μ_x are transitions which have just been executed. However, the latter part of such an interpretation may be somewhat misleading since it may happen that although (M,Q) is a reachable marking and $v \in \mu_x$, neither $(M \setminus v^\bullet, Q \cup \{v\})$ nor $((M \setminus v^\bullet) \cup {}^\bullet v, Q)$ are reachable from the entry marking of Ω; it may even happen that v is a dead transition, whose sole role is to force the factorisability of Ω. This is illustrated in figure 39 where the operator on the left hand side is factorisable, the transition v_2 is dead from the entry marking, and if we drop this dead (hence supposedly useless) transition the operator becomes non-factorisable since after the execution of v_1 and v_3, leading to the marking $(\{i,x\},\emptyset)$, the internal place i may no longer be factored out. It may easily be checked that all PBC operator boxes introduced in section 4.3.5 are factorisable. However, not all pure static boxes are factorisable, as illustrated on the right hand side of figure 39. For in the marking obtained from the entry marking after the execution of v_1 and v_3, the left token can be factored out using v_3 and the right token can be factored out using v_2, but the middle token is orphaned.
To define the domain of application of an sos-operator box Ω, we first extend the notion of a factorisation to tuples of static and dynamic boxes $\vec{\Sigma}$; the *factorisation* of $\vec{\Sigma}$ is $\mu = (\mu_e,\mu_d,\mu_x,\mu_s)$ where, for $\delta \in \{e,x,s\}$, $\mu_\delta = \{v \mid \Sigma_v \in \mathsf{Box}^\delta\}$, and $\mu_d = \{v \mid \Sigma_v \in \mathsf{Box}^d \setminus (\mathsf{Box}^e \cup \mathsf{Box}^x)\}$. The *domain* of application of Ω, denoted by dom_Ω, is then the set comprising every Ω–tuple of static and dynamic boxes $\vec{\Sigma}$ whose factorisation belongs to fact_Ω where fact_Ω is the set of all the factorisations of all the complex markings reachable from the entry marking of Ω, including the exit marking, as well as the only factorisation $(\emptyset,\emptyset,\emptyset,T_\Omega)$ of the empty marking of Ω.
Figure 40 shows an sos-operator box Ω_{re} which will serve as a running example in this section. Clearly, Ω_{re} satisfies (C1) and (C2). It is also factorisable which can be checked by inspecting all the complex markings reachable from the entry marking. E.g.: $({}^\circ\Omega_{re},\emptyset) = (\{s_1,s_2\},\emptyset)$ has a unique factorisation, $(\{v_1,v_2\},\emptyset,\emptyset,\{v_3\})$, and the reach-

able complex marking $(\{s_1\},\{v_2\})$ has a unique factorisation $(\{v_1\},\{v_2\},\emptyset,\{v_3\})$. In all, $\mathsf{fact}_{\Omega_{re}}$ comprises 13 factorisations. Figure 40 shows also an Ω_{re}-tuple of boxes, $\vec{\Upsilon} = (\Upsilon_{v_1},\Upsilon_{v_2},\Upsilon_{v_3})$ whose factorisation, $(\{v_1\},\{v_2\},\emptyset,\{v_3\})$, belongs to $\mathsf{fact}_{\Omega_{re}}$, hence $\vec{\Upsilon}$ is a tuple in the domain of application of the sos-operator box Ω_{re}. The box $\Omega_{re}(\vec{\Upsilon})$ is also shown in figure 40.

5.2 Structured operational semantics of boxes

Let Ω be an sos-operator box and $\vec{\Sigma}$ be an Ω–tuple in its domain. To formalise the operational semantics of the compositionally defined box $\Omega(\vec{\Sigma})$, we use the notation

$$(\Omega \,:\, \vec{\Sigma}) \xrightarrow{\;U\;} (\Omega \,:\, \vec{\Theta}) \tag{20}$$

to mean that the boxes $\vec{\Sigma}$ can individually make moves which, when combined, yield step U and lead to boxes $\vec{\Theta}$.

By definition, this will be the case whenever U is a set of transitions of $\Omega(\vec{\Sigma})$ and, for every transition v in Ω, $U \cap \mathsf{T}^{v}_{\mathsf{new}} = \{(v,\alpha_1)\lhd U_1,\ldots,(v,\alpha_k)\lhd U_k\}$ is a set[12] of transitions such that

$$\Sigma_v\,[\,U_1+\cdots+U_k\,\rangle\,\Theta_v. \tag{21}$$

For example, the boxes in figure 40 admit a move for $\vec{\Sigma} = \vec{\Upsilon}$, $\vec{\Theta} = (\underline{\Upsilon}_{v_1},\underline{\Upsilon}_{v_2},\Upsilon_{v_3})$ and $U = \{w_1,w_3\}$, which follows from: $U \cap \mathsf{T}^{v_1}_{\mathsf{new}} = \{w_1\} = \{(v_1,f)\lhd\{t_{11},t_{12}\}\}$, $U \cap \mathsf{T}^{v_2}_{\mathsf{new}} = \{w_3\} = \{(v_2,d)\lhd\{t_{22}\}\}$, $U \cap \mathsf{T}^{v_3}_{\mathsf{new}} = \emptyset$, $\Upsilon_{v_1}\,[\{t_{11},t_{12}\}\rangle\,\Theta_{v_1}$, $\Upsilon_{v_2}\,[\{t_{22}\}\rangle\,\Theta_{v_2}$ and $\Upsilon_{v_3}\,[\emptyset\rangle\,\Theta_{v_3}$.

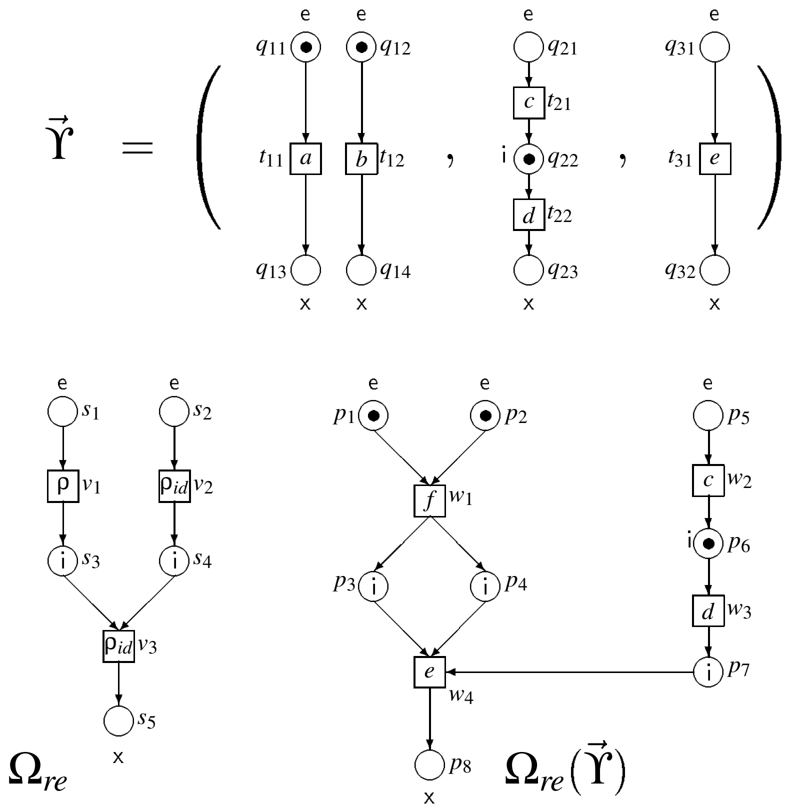
It can be shown that the multisets U_i in (21) are always sets of mutually independent transitions of Σ_v, and that allowing U to be a multiset rather than a set would not add any new moves (20) since $\mathsf{T}^{v}_{\mathsf{new}} \cap \mathsf{T}^{w}_{\mathsf{new}} = \emptyset$ for $v \neq w$, and all the boxes in $\vec{\Sigma}$ are safe. Notice that the definition of the operational semantics does not involve the redo and skip transitions, which are not involved in net refinement but added afterwards. Instead of expressing the evolution in terms of transitions, it is possible to express it in terms of actions, through the labelling function $\lambda_{\Omega(\vec{\Sigma})}$ which returns multisets rather than sets since different transitions may have the same label.What now follows is the first part of the SOS rule for compositionally defined boxes.

Theorem 5.1 Let $\vec{\Sigma}$ be a tuple in the domain of an sos-operator box Ω. If $\vec{\Theta}$ and U are as in (20), then $\vec{\Theta} \in \mathsf{dom}_{\Omega}$ and $\Omega(\vec{\Sigma})\,[U\rangle\,\Omega(\vec{\Theta})$. □

The converse does not in general hold true, however. For consider the tuple of boxes $\vec{\Theta} = (\underline{\Upsilon}_{v_1},\underline{\Upsilon}_{v_2},\Upsilon_{v_3})$. Then $M_{\Omega_{re}(\vec{\Theta})} = \{p_3,p_4,p_7\}$ and so $\Omega_{re}(\vec{\Theta})\,[\{w_4\}\rangle$, yet no move (20) is possible for $\Omega = \Omega_{re}$, $\vec{\Sigma} = \vec{\Theta}$ and a non-empty U. This is so because, when composing the nets, the tokens contributed by $\underline{\Upsilon}_{v_1}$ and $\underline{\Upsilon}_{v_2}$ are inserted into the composed net in such a way that they could have been contributed by the third box as well. More precisely, we have $\Omega(\vec{\Theta}) = \Omega(\vec{\Psi})$, where $\vec{\Psi} = (\lfloor\Upsilon_{v_1}\rfloor,\lfloor\Upsilon_{v_2}\rfloor,\overline{\Upsilon}_{v_3})$ and:

$$(\Omega_{re} \,:\, \vec{\Psi}) \xrightarrow{\{w_4\}} (\Omega_{re} \,:\, \vec{\Delta}) \quad \text{for} \quad \vec{\Delta} = (\lfloor\Upsilon_{v_1}\rfloor,\lfloor\Upsilon_{v_2}\rfloor,\underline{\Upsilon}_{v_3}).$$

[12]Note that the notation $\mathsf{T}^{v}_{\mathsf{new}}$ was introduced in (17).



$$
\begin{array}{lcllcl}
p_1 & = & s_1 \lhd v_1 \lhd q_{11} & p_2 & = & s_1 \lhd v_1 \lhd q_{12} \\
p_3 & = & s_3 \lhd \{v_1 \lhd q_{13}, v_3 \lhd q_{31}\} & p_4 & = & s_3 \lhd \{v_1 \lhd q_{14}, v_3 \lhd q_{31}\} \\
p_5 & = & s_2 \lhd v_2 \lhd q_{21} & p_6 & = & v_2 \lhd q_{22} \\
p_7 & = & s_4 \lhd \{v_2 \lhd q_{23}, v_3 \lhd q_{31}\} & p_8 & = & s_5 \lhd v_3 \lhd q_{33} \\
w_1 & = & (v_1, f) \lhd \{t_{11}, t_{12}\} & w_2 & = & (v_2, c) \lhd t_{21} \\
w_3 & = & (v_2, d) \lhd t_{22} & w_4 & = & (v_3, e) \lhd t_{31}
\end{array}
$$

Figure 40: Boxes of the running example ($\rho = \rho_{id} \backslash \{(\{a\}, a), (\{b\}, b)\} \cup \{(\{a, b\}, f)\}$).

Thus the markings in a tuple of boxes $\vec{\Sigma}$ may need to be rearranged before attempting to derive a move which is admitted by the composition $\Omega(\vec{\Sigma})$. Such a rearrangement is formalised using a similarity relation $\equiv_\Omega$ on Ω–tuples of boxes.[13]

Let Ω be an sos-operator box and $\vec{\Sigma}$ and $\vec{\Theta}$ be Ω–tuples of static and dynamic boxes whose factorisations are respectively μ and κ. Then $\vec{\Sigma} \equiv_\Omega \vec{\Theta}$ if μ and κ are factorisations of the same complex marking of Ω, $\lfloor\vec{\Sigma}\rfloor = \lfloor\vec{\Theta}\rfloor$ and $\Sigma_v = \Theta_v$, for every $v \in \mu_d = \kappa_d$. It is clear that $\equiv_\Omega$ is an equivalence relation. One can further strengthen this by showing that it is closed in the domain of Ω, and that it relates tuples which yield the same boxes through refinement, i.e., if $\vec{\Sigma} \in \mathsf{dom}_\Omega$ and $\vec{\Sigma} \equiv_\Omega \vec{\Theta}$, then $\vec{\Theta} \in \mathsf{dom}_\Omega$ and $\Omega(\vec{\Sigma}) = \Omega(\vec{\Theta})$. Moreover, if $\vec{\Sigma}, \vec{\Theta} \in \mathsf{dom}_\Omega$, $\lfloor\vec{\Sigma}\rfloor = \lfloor\vec{\Theta}\rfloor$ and $\Omega(\vec{\Sigma}) = \Omega(\vec{\Theta})$, then $\vec{\Sigma} \equiv_\Omega \vec{\Theta}$. For example, $(\underline{\Upsilon}_{v_1}, \underline{\Upsilon}_{v_2}, \Upsilon_{v_3}) \equiv_{\Omega_{re}} (\lfloor\Upsilon_{v_1}\rfloor, \lfloor\Upsilon_{v_2}\rfloor, \overline{\Upsilon}_{v_3})$.

We now can formulate the second half of the SOS rule for boxes. Together with theorem 5.1, it means that for the class of sos-operator boxes, the standard step sequence semantics of compositionally defined nets obeys a variant of the SOS rule introduced originally for process algebras.

Theorem 5.2 Let $\vec{\Psi}$ be an Ω-tuple in the domain of an sos-operator box Ω such that $\Omega(\vec{\Psi})\ [U\rangle\ \Sigma$. Then there are $\vec{\Sigma}$ and $\vec{\Theta}$ in dom_Ω such that $\vec{\Sigma} \equiv_\Omega \vec{\Psi}$, $\Omega(\vec{\Theta}) = \Sigma$ and (20) holds. $\square$

Various important consequences may be derived from theorems 5.1 and 5.2. In particular, they imply that Requirements 1 and 2 are satisfied. That is, $\Omega(\vec{\Sigma})$ is a static or dynamic box, and if Σ is a net derivable from $\overline{\Omega(\vec{\Sigma})}$, then there is a tuple $\vec{\Theta}$ in the domain of Ω such that $\Sigma = \Omega(\vec{\Theta})$. As a summary, the SOS rule for boxes can be presented in the following way:

$$\boxed{\frac{(\Omega\ :\ \vec{\Theta}) \xrightarrow{U} (\Omega\ :\ \vec{\Psi})}{\Omega(\vec{\Sigma})\ [U\rangle\ \Sigma} \quad \vec{\Sigma} \equiv_\Omega \vec{\Theta}\ ,\ \Omega(\vec{\Psi}) = \Sigma}$$

The algebra of nets used by PBC can be accommodated in the meta-scheme presented above since PBC operator boxes are all sos-operator boxes.

5.3 A process algebra and its semantics

We now introduce a general algebra of process expressions, called the *box algebra*. The box algebra is a *meta-model* parameterised by two non-empty, disjoint, possibly infinite, sets of Petri nets: a set $\mathsf{ConstBox}$ of static and dynamic (possibly infinite) plain boxes, and a set OpBox of sos-operator boxes. The only assumption about the sos-operator boxes in OpBox and the static boxes in $\mathsf{ConstBox}$ is that they have disjoint sets of single root trees as their places and transitions. We then consider an algebra of process expressions over the signature $\mathsf{Const}^s \cup \mathsf{Const}^d \cup \{\overline{(.)}, \underline{(.)}\} \cup \{\mathsf{op}_\Omega \mid \Omega \in \mathsf{OpBox}\}$ where Const^s and Const^d are fixed sets of *static* and *dynamic constants* which will be modelled through the boxes in $\mathsf{ConstBox}$, $\overline{(.)}$ and $\underline{(.)}$ are two unary operators, and each op_Ω is a connective of the algebra indexed by an sos-operator box taken from the set

[13] Note that the situation we just discussed is a mirror image of the re-distribution of over- and underbars used in the inaction rules of the operational semantics of PBC.

OpBox. Moreover, there are two distinct disjoint subsets of Const^d, denoted by Const^e and Const^x, and respectively called the *entry* and *exit* constants. We will also use a fixed set Var of *process variables*. Although we use the symbols $\overline{(.)}$ and $\underline{(.)}$ to denote both mappings on boxes and process algebra connectives, it will always be clear from the context what is the intended interpretation.

We shall make use of four classes of process expressions corresponding to previously introduced classes of plain boxes: the *entry*, *dynamic*, *exit* and *static* expressions, denoted respectively by Expr^e, Expr^d, Expr^x and Expr^s. Collectively, we will refer to them as the *box expressions*. We will also use a counterpart of the notion of the factorisation of a tuple of boxes. For an sos-operator box Ω and an Ω-tuple of box expressions $\vec{D}$, the *factorisation* of $\vec{D}$ is $\mu = (\mu_e, \mu_d, \mu_x, \mu_s)$ where $\mu_\delta = \{v \mid D_v \in \mathsf{Expr}^\delta\}$, for $\delta \in \{e,x,s\}$, and $\mu_d = \{v \mid D_v \in \mathsf{Expr}^d \setminus (\mathsf{Expr}^e \cup \mathsf{Expr}^x)\}$. The syntax for the box expressions is given by:

$$\begin{array}{llclllll} \mathsf{Expr}^s & E & ::= & c^s & \mid & X & \mid & \mathsf{op}_\Omega(\vec{E}) \\ \mathsf{Expr}^e & F & ::= & c^e & \mid & \overline{E} & \mid & \mathsf{op}_\Omega(\vec{F}) \\ \mathsf{Expr}^x & G & ::= & c^x & \mid & \underline{E} & \mid & \mathsf{op}_\Omega(\vec{G}) \\ \mathsf{Expr}^d & H & ::= & c^d & \mid & F & \mid & G \mid \mathsf{op}_\Omega(\vec{H}) \end{array} \tag{22}$$

where $c^\delta \in \mathsf{Const}^\delta$, for $\delta \in \{e,x,s\}$, and $c^d \in \mathsf{Const}^d \setminus (\mathsf{Const}^e \cup \mathsf{Const}^x)$ are constants; $X \in \mathsf{Var}$ is a process variable; $\Omega \in \mathsf{OpBox}$ is an sos-operator box; and $\vec{E}$, $\vec{F}$, $\vec{G}$ and $\vec{H}$ are Ω–tuples of box expressions. These tuples have to satisfy some conditions determined by the domain of application of the net operator induced by Ω. More precisely, the factorisations of $\vec{E}$, $\vec{F}$ and $\vec{G}$ are respectively factorisations of the complex empty, entry and exit markings of Ω, and the factorisation of $\vec{H}$ is a factorisation of a complex marking reachable from the entry marking of Ω different from ${}^\circ\Omega$ and Ω°. The definition of the syntax is completed by assuming that, for every process variable $X \in \mathsf{Var}$, there is a unique defining equation, $X \stackrel{\mathrm{df}}{=} \mathsf{op}_\Omega(\vec{L})$, where $\Omega \in \mathsf{OpBox}$ is an sos-operator box and $\vec{L}$ is an Ω–tuple of process variables and static constants. Expressions not involving any connective op_Ω nor a process variable, will be referred to as *flat*.

As in the case of boxes, it is convenient to have a notation for turning a box expression D into a corresponding static expression $\lfloor D \rfloor$ which is obtained from D by removing all the occurrences of $\overline{(.)}$ and $\underline{(.)}$, and replacing every occurrence of each dynamic constant c by a (fixed) corresponding static constant $\lfloor c \rfloor$. The operators $\overline{(.)}$, $\underline{(.)}$ and $\lfloor . \rfloor$ can be applied elementwise to sets as well as tuples of expressions. The same will be true of the mapping box and relation $\equiv$ defined later on.

We will continue to use the boxes depicted in figure 40 in order to construct a simple yet illustrative algebra of process expressions. The *Do It Yourself (DIY) algebra* is based on two sets of boxes, $\mathsf{ConstBox} = \{\Phi_1, \Phi_{11}, \Phi_{12}\} \cup \{\Phi_2, \Phi_{21}, \Phi_{22}, \Phi_{23}\} \cup \{\Phi_3\}$ and $\mathsf{OpBox} = \{\Omega_{re}\}$, where $\lfloor \Phi_{ij} \rfloor = \Phi_i = \lfloor \Upsilon_{v_i} \rfloor$ (for all i and j), $M_{\Phi_{11}} = \{q_{11}, q_{14}\}$, $M_{\Phi_{12}} = \{q_{13}, q_{12}\}$ and $M_{\Phi_{2k}} = \{q_{2k}\}$ (for $k = 1,2,3$). The constants of the DIY algebra correspond to the boxes in $\mathsf{ConstBox}$: $\mathsf{Const}^e = \{c_{21}\}$, $\mathsf{Const}^s = \{c_1, c_2, c_3\}$, $\mathsf{Const}^x = \{c_{23}\}$ and $\mathsf{Const}^d = \{c_{11}, c_{12}, c_{21}, c_{22}, c_{23}\}$. Moreover, $\lfloor c_{ij} \rfloor = c_i$, for every dynamic constant c_{ij}. The syntax of the DIY algebra is obtained by instantiating (22) with concrete constants and operator introduced above. For example, the syntax for the static and entry expressions is given respectively by $E ::= c_1 \mid c_2 \mid c_3 \mid X \mid \mathsf{op}_{\Omega_{re}}(E,E,E)$

and $F ::= c_{21} \mid \overline{E} \mid \mathsf{op}_{\Omega_{re}}(F,F,E)$.

5.3.1 Denotational semantics

The denotational semantics of the box algebra is given in the form of a mapping box from box expressions to boxes, defined by induction on the structure of expressions. Constant expressions are mapped onto constant boxes of corresponding types, i.e., for every constant c and $\delta \in \{e,d,x,s\}$, $c \in \mathsf{Const}^{\delta} \Leftrightarrow \mathsf{box}(c) \in \mathsf{Box}^{\delta} \cap \mathsf{ConstBox}$. It is also assumed that, for every dynamic constant c, the underlying box is the same as for the corresponding static constant, i.e., $\lfloor \mathsf{box}(c) \rfloor = \mathsf{box}(\lfloor c \rfloor)$, and that for every (non-entry and non-exit) dynamic box Σ reachable from an initially marked constant box there is a corresponding dynamic constant c, i.e., $\mathsf{box}(c) = \Sigma$.

With each defining equation $X \stackrel{\mathrm{df}}{=} \mathsf{op}_{\Omega}(\vec{L})$, we associate an equation on boxes $X \stackrel{\mathrm{df}}{=} \Omega(\vec{\Delta})$ where $\Delta_v = L_v$ if L_v is a process variable (treated here as a box variable), and $\Delta_v = \mathsf{box}(L_v)$ if L_v is a static constant. This creates a system of *equations on boxes* of the following form (one equation for every variable X in Var):

$$X \;\stackrel{\mathrm{df}}{=}\; \Omega_X(\vec{\Delta}_X). \tag{23}$$

On the right-hand side, Ω_X is an sos-operator box with the empty marking, and $\vec{\Delta}_X$ is an Ω_X-tuple whose elements are either recursion variables or plain boxes with empty markings. Now, given a mapping $\mathsf{sol} : \mathsf{Var} \to \mathsf{Box}^s$ assigning a static box to every variable in Var, we will denote by $\vec{\Delta}_X[\mathsf{sol}]$ the Ω_X-tuple obtained from $\vec{\Delta}_X$ by replacing each variable Y by $\mathsf{sol}(Y)$. Then, a *solution* of the system of equations (23) is an assignment sol such that, for every variable $X \in \mathsf{Var}$,

$$\mathsf{sol}(X) \;=\; \Omega_X(\vec{\Delta}_X[\mathsf{sol}]) \qquad \text{(where `$=$' denotes \textit{equality} on nets).}$$

It turns out that the system of net equations (23) always has at least one solution and that all solutions have the same behaviour since the corresponding static boxes have lbts-isomorphic transition systems. All the solutions can be obtained as limits of successive approximations starting from stop-boxes (see [6, 28]); such a result is possible due to the special way in which the names of the places and transitions were constructed in the definition of net refinement. In particular, one can define an inclusion order relation on boxes based on the concrete place and transition tree names. Then it can be shown that there is always the maximal solution of (23) for whom a closed form can be found in [6, 16].

We then fix *any* solution $\mathsf{sol} : \mathsf{Var} \to \mathsf{Box}^s$ of the recursive system (23) (for instance, the maximal one) and define, for every process variable X, $\mathsf{box}(X) = \mathsf{sol}(X)$.

The definition of box is completed by considering all the remaining static and dynamic expressions. For every box expression $\mathsf{op}_{\Omega}(\vec{D})$ and every static expression E, $\mathsf{box}(\mathsf{op}_{\Omega}(\vec{D})) = \Omega(\mathsf{box}(\vec{D}))$, $\mathsf{box}(\overline{E}) = \overline{\mathsf{box}(E)}$ and $\mathsf{box}(\underline{E}) = \underline{\mathsf{box}(E)}$. The semantical mapping always returns a box consistent with the type of a box expression it was applied to; hence we have captured syntactically the property of being a static, dynamic, entry or exit box.

Theorem 5.3 For every box expression D, $\mathsf{box}(D)$ is a static or dynamic box. Moreover, for every $\delta \in \{e,d,x,s\}$, $D \in \mathsf{Expr}^{\delta} \Leftrightarrow \mathsf{box}(D) \in \mathsf{Box}^{\delta}$. $\square$

In the case of the DIY algebra, we define the box mapping by setting, for every static constant c_i, $\mathsf{box}(c_i) = \Phi_i$, and for every dynamic constant c_{ij}, $\mathsf{box}(c_{ij}) = \Phi_{ij}$. Other than that, we follow the general definitions. E.g., the box in figure 40 can be derived thus: $\mathsf{box}(\mathsf{op}_{\Omega_{re}}(\overline{c_1}, c_{22}, c_3)) = \Omega_{re}(\mathsf{box}(\overline{c_1}), \mathsf{box}(c_{22}), \mathsf{box}(c_3)) = \Omega_{re}(\overline{\mathsf{box}(c_1)}, \Phi_{22}, \Phi_3)$ $= \Omega_{re}(\overline{\Phi_1}, \Phi_{22}, \Phi_3) = \Omega_{re}(\Upsilon_{v_1}, \Upsilon_{v_2}, \Upsilon_{v_3}) = \Omega_{re}(\vec{\Upsilon})$.

5.3.2 Structural similarity relation on expressions

To facilitate the introduction of the inaction rule we define a structural similarity relation on box expressions, $\equiv$. It provides a partial structural identification of the box expressions with the same denotational semantics, as it is defined as the least equivalence relation on box expressions such that the following hold.

- For all flat expressions D and H satisfying $\mathsf{box}(D) = \mathsf{box}(H)$, and all equations $X \stackrel{\text{df}}{=} \mathsf{op}_\Omega(\vec{L})$,

$$\boxed{D \equiv H \qquad X \equiv \mathsf{op}_\Omega(\vec{L}).} \tag{24}$$

- For every sos-operator box Ω in OpBox and all factorisations μ and κ of respectively ${}^\circ\Omega$ and Ω°:

$$\boxed{\mathsf{op}_\Omega(\vec{D}) \equiv \overline{\mathsf{op}_\Omega(\vec{H})} \qquad \mathsf{op}_\Omega(\vec{J}) \equiv \underline{\mathsf{op}_\Omega(\vec{H})}.} \tag{25}$$

  where $\vec{D}$, $\vec{J}$ and $\vec{H}$ are Ω-tuples of expressions such that, for every $v \in T_\Omega$, $D_v = \overline{H_v}$ if $v \in \mu_e$ and $D_v = H_v$ otherwise; and $J_v = \underline{H_v}$ if $v \in \kappa_x$ and $J_v = H_v$ otherwise.

- For every sos-operator box Ω in OpBox, for every complex marking reachable from the entry marking of Ω, different from ${}^\circ\Omega$ and Ω°, and for every pair of different factorisations μ and κ of that marking,

$$\boxed{\mathsf{op}_\Omega(\vec{D}) \equiv \mathsf{op}_\Omega(\vec{H})} \tag{26}$$

  where $\vec{D}$ and $\vec{H}$ are Ω-tuples of expressions for which there is an Ω-tuple of expressions $\vec{C}$ such that, for every $v \in T_\Omega$, $D_v = \overline{C_v}$ if $v \in \mu_e$, $D_v = \underline{C_v}$ if $v \in \mu_x$ and $D_v = C_v$ otherwise; and $H_v = \overline{C_v}$ if $v \in \kappa_e$, $H_v = \underline{C_v}$ if $v \in \kappa_x$ and $H_v = C_v$ otherwise.

- For all static expressions E and F, for every sos-operator box Ω in OpBox, and for all Ω-tuples of expressions $\vec{D}$ and $\vec{H}$ with factorisations in fact_Ω:

$$\boxed{\frac{E \equiv F}{\overline{E} \equiv \overline{F}} \qquad \frac{E \equiv F}{\underline{E} \equiv \underline{F}} \qquad \frac{\vec{D} \equiv \vec{H}}{\mathsf{op}_\Omega(\vec{D}) \equiv \mathsf{op}_\Omega(\vec{H})}} \tag{27}$$

The tuple $\vec{C}$ used in the formulation of (26) intuitively corresponds to the ‘common’ part of $\vec{D}$ and $\vec{H}$. The structural similarity relation is closed in the domain of box expressions[14] and preserves the types of expressions it relates.

[14] That is, whenever a box expression can match one side of a rule, then it is also guaranteed that the other side is a box expression too. Thus the rules can be thought of as well formed term rewriting rules. Similar comment applies to the rules of the structured operational semantics.

Theorem 5.4 Let D and H be box expressions such that $D \equiv H$. Then $\mathsf{box}(D) = \mathsf{box}(H)$ and, for every $\delta \in \{e,d,x,s\}$, $D \in \mathsf{Expr}^{\delta} \Leftrightarrow H \in \mathsf{Expr}^{\delta}$. $\square$

Thus $\equiv$ is a sound equivalence notion from the point of view of the denotational semantics of box expressions. It is also complete in the sense that $\mathsf{box}(D) = \mathsf{box}(H)$ implies $D \equiv H$ provided that $\lfloor D \rfloor \equiv \lfloor H \rfloor$. The latter condition cannot be left out and, in terms of the DIY algebra, a counterexample is provided by the expressions $D = \overline{X}$ and $H = \overline{Y}$, where X and Y are variables defined by $X \stackrel{\mathrm{df}}{=} \Omega_{re}(c_1, c_1, X)$ and $Y \stackrel{\mathrm{df}}{=} \Omega_{re}(c_1, c_1, Y)$.

The DIY algebra gives rise to five specific rules for the structural equivalence relation (we omit here their symmetric, hence redundant, counterparts). The first two are derived from (24): $\overline{c_2} \equiv c_{21}$ and $\underline{c_2} \equiv c_{23}$. The third and fourth are derived from (25): $\mathsf{op}_{\Omega_{re}}(\overline{E}, \overline{F}, G) \equiv \overline{\mathsf{op}_{\Omega_{re}}(E, F, G)}$ and $\mathsf{op}_{\Omega_{re}}(E, F, \underline{G}) \equiv \underline{\mathsf{op}_{\Omega_{re}}(E, F, G)}$. Finally, there is a single instance of (26): $\mathsf{op}_{\Omega_{re}}(\underline{E}, \underline{F}, G) \equiv \mathsf{op}_{\Omega_{re}}(E, F, \overline{G})$. An application of these rules is illustrated by the following derivation tree:

$$\dfrac{\dfrac{\underline{c_1} \equiv \underline{c_1} \qquad c_{23} \equiv \underline{c_2} \qquad c_3 \equiv c_3}{\mathsf{op}_{\Omega_{re}}(\underline{c_1}, c_{23}, c_3) \equiv \mathsf{op}_{\Omega_{re}}(\underline{c_1}, \underline{c_2}, c_3)} \qquad \mathsf{op}_{\Omega_{re}}(\underline{c_1}, \underline{c_2}, c_3) \equiv \mathsf{op}_{\Omega_{re}}(c_1, c_2, \overline{c_3})}{\mathsf{op}_{\Omega_{re}}(\underline{c_1}, c_{23}, c_3) \equiv \mathsf{op}_{\Omega_{re}}(c_1, c_2, \overline{c_3})}$$

5.3.3 Operational semantics

The derivation system we shall define has moves of the form $D \xrightarrow{\Gamma} H$, where D and H are expressions and Γ is a finite multiset of labels. Formally, we define a ternary relation $\longrightarrow$ which is the least relation comprising all the triples (D, Γ, H), where D and H are box expressions and $\Gamma \in \mathsf{mult}(\mathsf{Lab} \cup \{\mathsf{skip}, \mathsf{redo}\})$, such that the following hold (note that we write $D \xrightarrow{\Gamma} H$ instead of $(D, \Gamma, H) \in \longrightarrow$).

- For every static expression E:

$$\boxed{\overline{E} \xrightarrow{\{\mathsf{skip}\}} \underline{E} \qquad \underline{E} \xrightarrow{\{\mathsf{redo}\}} \overline{E}.} \tag{28}$$

- For all box expressions D, J and H:

$$\boxed{\dfrac{D \equiv H}{D \xrightarrow{\emptyset} H} \qquad \dfrac{D \xrightarrow{\emptyset} J \xrightarrow{\Gamma} H}{D \xrightarrow{\Gamma} H} \qquad \dfrac{D \xrightarrow{\Gamma} J \xrightarrow{\emptyset} H}{D \xrightarrow{\Gamma} H}} \tag{29}$$

- For every flat expression D and a non-empty step of transitions U enabled by $\mathsf{box}(D)$, there is a flat expression H such that $\mathsf{box}(D)\ [U\rangle\ \mathsf{box}(H)$, $\lambda_{\mathsf{box}(D)}(U) = \Gamma$ and:

$$\boxed{D \xrightarrow{\Gamma} H.} \tag{30}$$

- For every $\Omega \in \mathsf{OpBox}$, and all Ω-tuples $\vec{D}$ and $\vec{H}$ of expressions:

$$\boxed{\dfrac{\forall v \in T_\Omega\colon D_v \xrightarrow{\Gamma_v^1 + \cdots + \Gamma_v^{k_v}} H_v}{\mathsf{op}_\Omega(\vec{D}) \xrightarrow{\sum_{v\in T_\Omega}\{\alpha_v^1\} + \cdots + \{\alpha_v^{k_v}\}} \mathsf{op}_\Omega(\vec{H})} \;\; (\Gamma_v^i, \alpha_v^i) \in \lambda_\Omega(v)} \tag{31}$$

Notice that the only way to generate a skip or redo is through applying the rules (28), and possibly the rules (29) afterwards; thus, for example, if $D \xrightarrow{\Gamma} H$ and $\mathsf{skip} \in \Gamma$ then $\Gamma = \{\mathsf{skip}\}$. Notice also that $D \xrightarrow{\emptyset} H$ if and only if $D \equiv H$. A crucial property of the operational semantics is that it transforms a box expression into another box expression, and the move generated is a valid step for the corresponding boxes, and vice versa.

Theorem 5.5 Let D be a box expression. If $D \xrightarrow{\Gamma} H$ then H is a box expression such that $(\mathsf{box}(D))_{\mathsf{sr}}\,[\Gamma\rangle_{\mathsf{lab}}\,(\mathsf{box}(H))_{\mathsf{sr}}$. Conversely, if $(\mathsf{box}(D))_{\mathsf{sr}}\,[\Gamma\rangle_{\mathsf{lab}}\,\Sigma_{\mathsf{sr}}$ then there is a box expression H such that $\mathsf{box}(H) = \Sigma$ and $D \xrightarrow{\Gamma} H$. $\square$

In the DIY algebra, the operational semantics of flat expressions is given by: $\overline{c_1} \xrightarrow{\{a\}} c_{12}$, $\overline{c_1} \xrightarrow{\{b\}} c_{11}$, $\overline{c_1} \xrightarrow{\{a,b\}} \underline{c_1}$, $c_{11} \xrightarrow{\{a\}} \underline{c_1}$, $c_{12} \xrightarrow{\{b\}} \underline{c_1}$, $\overline{c_2} \xrightarrow{\{c\}} c_{22}$, $c_{21} \xrightarrow{\{c\}} c_{22}$, $c_{22} \xrightarrow{\{d\}} \underline{c_2}$, $c_{22} \xrightarrow{\{d\}} c_{23}$ and $\overline{c_3} \xrightarrow{\{e\}} \underline{c_3}$. The inference rule for the only operator box can be formulated in the following way:

$$\dfrac{D_1 \xrightarrow{k\cdot\{a,b\}+\Gamma_1} H_1 \quad , \quad D_2 \xrightarrow{\Gamma_2} H_2 \quad , \quad D_3 \xrightarrow{\Gamma_3} H_3}{\mathsf{op}_{\Omega_{re}}(D_1,D_2,D_3) \xrightarrow{k\cdot\{f\}+\Gamma_1+\Gamma_2+\Gamma_3} \mathsf{op}_{\Omega_{re}}(H_1,H_2,H_3)} \; a,b \notin \Gamma_1$$

An application of these rules is shown below (where $\mathsf{op}_{\Omega_{re}}$ is denoted by op):

$$\overline{\mathsf{op}(c_1,c_2,c_3)} \xrightarrow{\{c,f\}} \mathsf{op}(\underline{c_1},c_{22},c_3)$$

$$\overbrace{\overline{\mathsf{op}(c_1,c_2,c_3)} \xrightarrow{\emptyset} \mathsf{op}(\overline{c_1},\overline{c_2},c_3) \qquad\qquad \mathsf{op}(\overline{c_1},\overline{c_2},c_3) \xrightarrow{\{c,f\}} \mathsf{op}(\underline{c_1},c_{22},c_3)}$$

$$\overbrace{\overline{c_1} \xrightarrow{\{a,b\}} \underline{c_1} \qquad \overline{c_2} \xrightarrow{\{c\}} c_{22} \qquad c_3 \xrightarrow{\emptyset} c_3}$$

The consistency between the denotational and operational semantics of box expressions will be expressed by making a statement about the transition systems they generate (see also sections 3.2.1 and 4.1.3). With each static or dynamic expression D, we associate two *transition systems*, the labelled transition system lbts_D and the reduced labelled transition system lbts_D^{rdc}. Both are defined as in section 3, with obvious modifications required by a more general framework. We then can state a fundamental result that the operational and denotational semantics of a box expression capture the same behaviour, in the strongest sense of this word.

Theorem 5.6 For every box expression D, lbts_D^{rdc} and $\mathsf{lbts}_{\mathsf{box}(D)}$ are isomorphic transition systems. $\square$

Moreover, the mapping $\mathsf{iso} = \{(\nu, \mathsf{box}(H)) \mid \nu$ is a node in lbts_D^{rdc} and $H \in \nu\}$ is an isomorphism for lbts_D^{rdc} and $\mathsf{lbts}_{\mathsf{box}(D)}$. One can also show that lbts_D and lbts_D^{rdc} are strongly equivalent transition systems.
In the DIY algebra, the last theorem can be illustrated by taking the expression (again, op denotes $\mathsf{op}_{\Omega_{re}}$) $D = \mathsf{op}(\overline{c_1}, c_{22}, c_3)$ and the corresponding box shown in figure 40. Figure 41 depicts lbts_D^{rdc} and $\mathsf{lbts}_{\mathsf{box}(D)}$. The nodes of lbts_D^{rdc} are:
$\nu_0 = \{\mathsf{op}(\overline{c_1}, c_{22}, c_3)\}$, $\nu_1 = \{\overline{\mathsf{op}(c_1, c_2, c_3)}, \mathsf{op}(\overline{c_1}, \overline{c_2}, c_3), \mathsf{op}(\overline{c_1}, c_{21}, c_3)\}$,
$\nu_2 = \{\mathsf{op}(\underline{c_1}, \overline{c_2}, c_3), \mathsf{op}(\underline{c_1}, c_{21}, c_3)\}$, $\nu_3 = \{\mathsf{op}(\underline{c_1}, c_{22}, c_3)\}$,
$\nu_4 = \{\mathsf{op}(\overline{c_1}, c_{23}, c_3)$, $\mathsf{op}(\overline{c_1}, \underline{c_2}, c_3)\}$, $\nu_5 = \{\mathsf{op}(\underline{c_1}, \underline{c_2}, c_3), \mathsf{op}(\underline{c_1}, c_{23}, c_3), \mathsf{op}(c_1, c_2, \overline{c_3})\}$
and $\nu_6 = \{\underline{\mathsf{op}(c_1, c_2, c_3)}, \mathsf{op}(c_1, c_2, \underline{c_3})\}$.
The nodes of $\mathsf{lbts}_{\mathsf{box}(D)}$ are such that: $M_{\Sigma_0} = \{p_1, p_2, p_6\}$, $M_{\Sigma_1} = \{p_1, p_2, p_5\}$, $M_{\Sigma_2} = \{p_3, p_4, p_5\}$, $M_{\Sigma_3} = \{p_3, p_4, p_6\}$, $M_{\Sigma_4} = \{p_1, p_2, p_7\}$, $M_{\Sigma_5} = \{p_3, p_4, p_7\}$ and $M_{\Sigma_6} = \{p_8\}$.

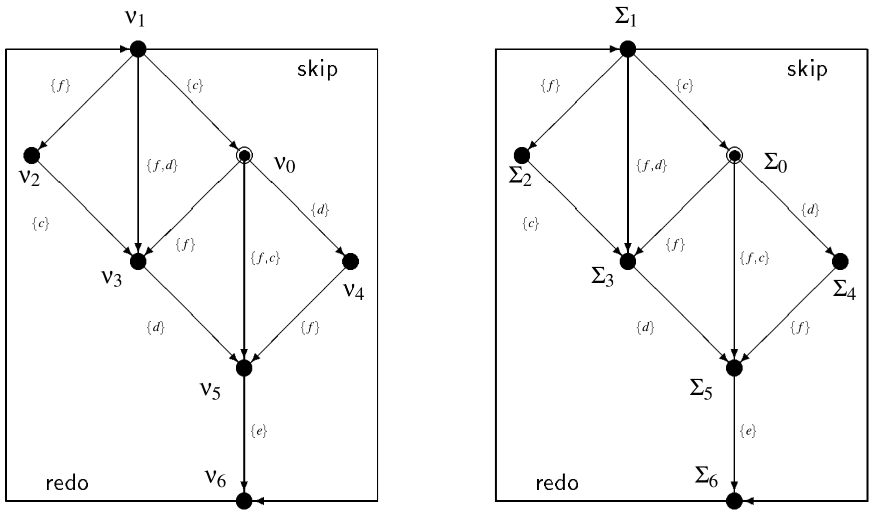


Figure 41: Two isomorphic transition systems, lbts_D^{rdc} and $\mathsf{lbts}_{\mathsf{box}(D)}$.

We now return to the factorisability property. Consider again the non-factorisable operator box shown on the right of figure 39. To see that this operator does not have a complete operational semantics in the style outlined above, try defining a quaternary operator $\mathsf{op}(E_1, E_2, E_3, E_4)$ from it, with four arguments (where E_1 – E_4 are supposed to describe the same flow of control as $v_1 - v_4$ of the operator) and the following set of equations:

$$\begin{array}{rcl}
\overline{\mathsf{op}(E_1, E_2, E_3, E_4)} & \equiv & \mathsf{op}(\overline{E_1}, \overline{E_2}, E_3, E_4), \\
\mathsf{op}(\underline{E_1}, \overline{E_2}, E_3, E_4) & \equiv & \mathsf{op}(E_1, \overline{E_2}, \overline{E_3}, E_4), \\
\mathsf{op}(\underline{E_1}, \underline{E_2}, E_3, E_4) & \equiv & \mathsf{op}(E_1, E_2, \overline{E_3}, \overline{E_4}), \\
\mathsf{op}(E_1, \underline{E_2}, \underline{E_3}, E_4) & \equiv & \mathsf{op}(E_1, E_2, \underline{E_3}, \overline{E_4}), \\
\mathsf{op}(E_1, E_2, \underline{E_3}, \underline{E_4}) & \equiv & \underline{\mathsf{op}(E_1, E_2, E_3, E_4)}.
\end{array}$$

At first sight, the equations seem to describe the control flow within the operator box. However, if we consider the expression $E = \mathsf{op}(\ a\ , (b; b')\ , (c; c')\ , d\ )$ then the box corresponding to the expression $\overline{E}$ can execute the sequence of labels, $abcb'd$, which cannot be derived using the above set of equations (essentially, the third equation cannot be applied in this case).

5.4 PBC, CCS, TCSP and COSY

The syntax of static and dynamic PBC expressions was given in sections 3.1 and 3.3.4. It is not difficult to see that the syntax conforms to that of the general box algebra in (22), after making some natural simplifications and adjustments, such as changing the mode of application of the operators. The only operator which requires some care is iteration, which strictly speaking is a 6-ary operator; however, due to its fully symmetric form one can treat it as though it were a ternary operator, both in the syntax definition and later, in the definitions of the rules of the operational semantics. Notice that all the operators modelling the various constructs of PBC are sos-operator boxes, and that the set of constants comprises all the basic actions and the expressions used in the modelling of data variables; in particular, $\mathsf{Const}^s = \mathsf{Lab}_{\mathrm{PBC}} \cup \{[z_{()}] \mid z \text{ is a data variable}\}$. The semantical mapping from the PBC expressions to boxes, $\mathsf{box}_{\mathrm{PBC}}$, has been defined following the generic definition introduced in this section. The equation system used in section 3 to define the structural equivalence on PBC expressions is an instance of that defined for the general box algebra, with some notational modifications and simplifications. In particular, IN1 and IN2 follow from $\equiv$ being an equivalence relation; ILN and IRN follow from (29); AR, DAT1, DAT2 and DAT3 follow from (30); IPAR1, IC1L, IPAR2 and IC2L follow from (25); IS2, IIT2a, IIT2b and IIT2c follow from (26); PAR, CL, CR, SL,SR, SY1, SY2, IT1, IT2 and IT3 follow from (31); and IREC follows from (24) and (27). Hence we may conclude that the operational and denotational semantics defined for the PBC algebra in the previous sections are consistent, in the sense of theorem 5.6.

In the rest of this section we shall outline how CCS [31, 32], TCSP [25] and COSY [26] could be treated within the general compositionality framework provided by the box algebra. In what follows, by a *simple operator* we will mean a two-place one-transition operator box as shown in section 4.3.4. We will denote such an operator box by $\Omega{:}\rho$, where ρ is the relabelling of its only transition.

To model CCS we assume that $\mathsf{Lab} = Act \cup \{\tau\}$ is the set of CCS labels and $\widehat{\ }: Act \to Act$ is a bijection on Act satisfying $\widehat{\widehat{a}} = a$ and $\widehat{a} \neq a$, for all $a \in Act$. We then define five simple operators: $\Omega{:}restr(A)$, $\Omega{:}relab(h)$, $\Omega{:}left$, $\Omega{:}right$ and $\Omega{:}syn(CCS)$, where A is a set of labels and h is a mapping on labels $h : \mathsf{Lab} \to \mathsf{Lab}$ commuting with $\widehat{(.)}$, defined by the relabellings: $restr(A) = \{(\{a\}, a) \mid a \in (\mathsf{Lab} \setminus A)\}$, $relab(h) = \{(\{a\}, h(a)) \mid a \in \mathsf{Lab}\}$, $left = \{(\{a\}, a^L) \mid a \in \mathsf{Lab}\}$, $right = \{(\{a\}, a^R) \mid a \in \mathsf{Lab}\}$ and

$$syn(CCS) = \{(\{a^L, \widehat{a}^R\}, \tau) \mid a \in Act\} \cup \{(\{a^L\}, a), (\{a^R\}, a) \mid a \in \mathsf{Lab}\}.$$

Note that it is assumed that Lab is extended by the labels a^L and a^R, for $a \in \mathsf{Lab}$, but neither a^L nor a^R are allowed in the syntax of CCS expressions; they are a mere artifact used to model correctly the semantics of CCS. The translation ϕ_{CCS} from CCS processes into box algebra expressions, using the CCS syntax (1), is given by $\phi_{CCS}(\mathsf{nil}) = \mathsf{stop}$

and:

$$\begin{array}{lcl}
\phi_{CCS}(E\backslash a) & = & \mathsf{op}_{\Omega:restr(\{a,\hat{a}\})}(\phi_{CCS}(E)) \\
\phi_{CCS}(\alpha.E) & = & \mathsf{op}_{\Omega_;}(\mathsf{base}_\alpha, \phi_{CCS}(E)) \\
\phi_{CCS}(E+F) & = & \mathsf{op}_{\Omega_{\square}}(\phi_{CCS}(E), \phi_{CCS}(F)) \\
\phi_{CCS}(E[h]) & = & \mathsf{op}_{\Omega:relab(h)}(\phi_{CCS}(E)) \\
\phi_{CCS}(E|F) & = & \mathsf{op}_{\Omega:syn(CCS)}(\mathsf{op}_{\Omega_{\|}}(\mathsf{op}_{\Omega:left}(\phi_{CCS}(E)), \mathsf{op}_{\Omega:right}(\phi_{CCS}(F)))).
\end{array}$$

TCSP and COSY employ synchronisation mechanisms different from those used in CCS and PBC. We shall briefly explain how concurrent composition and synchronisation used in these two models could be treated.

The concurrent composition in TCSP is a binary operator, $E\|_A F$, where $A \subseteq \mathsf{Lab}$ is a set of actions on which $\|_A$ enforces synchronisation. The resulting process can execute an action $a \in A$ if it can be executed simultaneously by the two component processes; this simultaneous execution is denoted by a. The actions outside A can be executed autonomously by the two component processes. Such a synchronisation discipline can be modelled similarly as in CCS. This time, however, we do not assume that the set of labels, Lab, has any special properties. The relevant fragment of the transformation ϕ_{TCSP} from TCSP expressions to box algebra expressions is modelled thus:

$$\phi_{TCSP}(E\|_A F) \ = \ \mathsf{op}_{\Omega:TCSP(A)}(\mathsf{op}_{\Omega_{\|}}(\mathsf{op}_{\Omega:left}(\phi_{TCSP}(E)), \mathsf{op}_{\Omega:right}(\phi_{TCSP}(F))))$$

where Ω:$TCSP(A)$ is a simple operator with the relabelling defined thus:

$$TCSP(A) \ = \ \{(\{a^L, a^R\}, a) \mid a \in A\} \cup \{(\{a^L\}, a), \{a^R\}, a) \mid a \in (\mathsf{Lab} \setminus A)\}.$$

As before, we assume that the set of labels is temporarily extended by a^L and a^R, for $a \in \mathsf{Lab}$, which are not in the TCSP syntax.

The concurrent composition in COSY is based on a multi-way synchronisation. Consider a path program

$$prog \ = \ \mathsf{program} \ \ path_1 \ \ \ldots \ \ path_n \ \ \mathsf{endprogram}$$

Here, $prog$ can execute a if it is executed simultaneously in all the paths $path_i$ in which a occurs. To model such a synchronisation mechanism, we first extend $\Omega_{\|}$ to n-ary ($n \geq 1$ is finite) parallel composition operator boxes, $\Omega_{\|}^n$, in the obvious way (see section 4.3.7). Let $A \subseteq \mathsf{Lab}$ be the set of labels occurring in program $prog$ and, for every $a \in A$, let $ix(a)$ be the set of all the indices i such that a occurs in the path $path_i$. Let $Index = \{(a, ix(a)) \mid a \in A\}$. Define a simple operator Ω:$COSY(Index)$ with the relabelling being given by:

$$COSY(Index) \ = \ \bigcup_{a \in A} \{(\{a^i \mid i \in ix(a)\}, a)\}$$

and n simple operators, Ω:ix_i, each with the relabelling $ix_i = \{(\{a\}, a^i) \mid a \in \mathsf{Lab}\}$. Then the relevant fragment of the transformation ϕ_{COSY} from COSY programs to box algebra expressions is:

$$\phi_{COSY}(prog) \ =$$
$$\mathsf{op}_{\Omega:COSY(Index)}(\mathsf{op}_{\Omega_{\|}^n}(\mathsf{op}_{\Omega:ix_1}(\phi_{COSY}(path_1)), \ldots, \mathsf{op}_{\Omega:ix_n}(\phi_{COSY}(path_n)))).$$

6 Concurrent Programming Languages

In this section, we discuss the use of process algebras in giving the semantics of concurrent programming languages. We define an example language, called *EL*, and use PBC as a semantic domain for *EL*. This also induces, by the Petri net semantics of PBC, a consistent Petri net semantics of *EL*. *EL* will be a shared data language, meaning that it is possible to express concurrent processes operating on a common set of variables. Alternatively, *EL* can be viewed as a modification of Dijkstra's guarded command language [18].

In section 6.1, we describe the syntax of *EL*. In section 6.2, we give its semantics in terms of PBC and thus, implicitly, in terms of Petri nets. More precisely, we define a mapping which associates with every *EL* program a PBC expression (and thus a net). One of the characteristic properties of this mapping is that it is compositional, i.e., it has the homomorphism property, by mapping operators of the programming language to operations on PBC expressions (and thus on nets). Section 6.3, finally, contains some discussion on possible applications.

6.1 Syntax of *EL*

The syntax of *EL* is specified in table 3. It is assumed that *Set* (the type of the variable x declared) denotes an arbitrary set, *BExpr* denotes a Boolean expression and *Expr* denotes an expression. We assume all the usual context restrictions, such as that if x and *Expr* occur in an assignment $x{:=}\mathit{Expr}$, then the type of *Expr* must be the same as that of x. The intuitive meaning of the brackets $\llbracket \ldots \rrbracket$ is that they enclose atomic actions; thus, for instance, $V(x)$, with a semaphore variable x, can be encoded as $\llbracket x{:=}x{+}1 \rrbracket$, and $P(x)$ can be encoded as $\mathbf{if}\ \llbracket\ (x{>}0)\,;(x{:=}x{-}1)\ \rrbracket\ \mathbf{fi}$. The syntactic entity *GC* is called a 'guarded command', and the Boolean expression in its first part is called its 'guard'. Note that choices **if** ... **fi** and loops **do** ... **od** may not start directly with other loops, since an inner loop is always guarded by a Boolean expression. As the reader will recall from sections 3.3.1 and 4.3.8, this excludes some undesirable semantic situations relating to generalised loops, and thus we may use an appropriate generalised loop construct of PBC to give the semantics of *EL*.

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \mathit{Block} \\
\mathit{Block} & ::= & \mathbf{begin}\ \mathit{Body}\ \mathbf{end} \\
\mathit{Body} & ::= & \mathit{Decl};\mathit{Body} \mid \mathit{Com} \\
\mathit{Decl} & ::= & \mathbf{var}\ x : \mathit{Set} \\
\mathit{Com} & ::= & \mathit{Block} \mid \mathit{Act} \mid \mathit{Com};\mathit{Com} \mid \mathit{Com}\|\mathit{Com} \mid \\
 & & \mathbf{if}\ \mathit{GC}\ \square\ \ldots\ \square\ \mathit{GC}\ \mathbf{fi} \mid \mathbf{do}\ \mathit{GC}\ \square\ \ldots\ \square\ \mathit{GC}\ \mathbf{od} \\
\mathit{Act} & ::= & \llbracket x{:=}\mathit{Expr} \rrbracket \\
\mathit{GC} & ::= & \mathit{GC};\mathit{Com} \mid \llbracket \mathit{BExpr} \rrbracket \mid \llbracket \mathit{BExpr};x{:=}\mathit{Expr} \rrbracket .
\end{array}
$$

Table 3: Syntax of *EL*.

6.2 Semantics of *EL*

Let P be an *EL* program fragment, which could be a program, a block, a body, a declaration, a command, etc. We will now define a mapping pbc_{EL} associating a PBC expression $\mathsf{pbc}_{EL}(P)$, and thus a box, $\mathsf{box}_{\mathrm{PBC}}(\mathsf{pbc}_{EL}(P))$, with P. We proceed by induction on the syntax of *EL*.

6.2.1 Programs, blocks and declarations

The main idea in describing a block – due to Milner [31, 32] – is to juxtapose (in parallel) the nets for its declarations and the net for its body, followed by termination action(s), to synchronise all matching data / command transitions, and then to restrict them in order to make local variables invisible outside the block. An example has already been described in section 2.8.

Assume that *Decl* equals $\mathbf{var}\, x : Set$. Then, by definition, $D_x = Set$ is the value domain of x. Moreover, we abbreviate:

$$\delta(Decl) = \{\, x_{kl} \mid k,l \in Set \cup \{\bullet\} \,\} \quad \text{and} \quad \tau(Decl) = \square_{k \in Set \cup \{\bullet\}}\, x_{k\bullet}$$

The set $\delta(Decl)$ is the set of all action particles that pertain to the declaration of x; this set is used for scoping. The expression $\tau(Decl)$ is a generalised choice (cf. section 3.3.3) of all action particles that would lead to the termination of the variable; this is used for termination of the block in which the declaration occurs. We define the semantics of blocks (and hence *EL* programs) as follows:

$$\begin{array}{lcl}
\mathsf{pbc}_{EL}(\mathbf{begin}\, Body\, \mathbf{end}) & = & \mathsf{pbc}_{EL}(Body) \\
\mathsf{pbc}_{EL}(Decl; Body) & = & [\,\delta(Decl) : (\,\mathsf{pbc}_{EL}(Decl) \,\|\, (\,\mathsf{pbc}_{EL}(Body); \tau(Decl)\,)\,] \\
\mathsf{pbc}_{EL}(\,\mathbf{var}\, x : Set\,) & = & (\,[\hat{x}_{()}] \;\square\; \{\hat{x}_{\bullet\bullet}\}\,).
\end{array}$$

Note that we have used both the generalised operator $[A : E]$ (section 3.3.3) and the data expression constant (section 3.3.2).

6.2.2 Command connectives

The semantics of the command connectives is defined in the following way:

$$\begin{array}{lcl}
\mathsf{pbc}_{EL}(Com_1; Com_2) & = & \mathsf{pbc}_{EL}(Com_1)\,;\,\mathsf{pbc}_{EL}(Com_2) \\
\mathsf{pbc}_{EL}(Com_1 \| Com_2) & = & \mathsf{pbc}_{EL}(Com_1) \,\|\, \mathsf{pbc}_{EL}(Com_2) \\
\mathsf{pbc}_{EL}(\,\mathbf{if}\, GC_1 \;\square\; \ldots \;\square\; GC_m\, \mathbf{fi}\,) & = & \mathsf{pbc}_{EL}(GC_1) \;\square\; \ldots \;\square\; \mathsf{pbc}_{EL}(GC_m) \\
\mathsf{pbc}_{EL}(\,\mathbf{do}\, GC_1 \;\square\; \ldots \;\square\; GC_m\, \mathbf{od}\,) & = & \langle\, (\mathsf{pbc}_{EL}(GC_1) \;\square\; \ldots \;\square\; \mathsf{pbc}_{EL}(GC_m)) \\
 & & \;\; * \;\mathsf{pbc}_{EL}([\![\neg B_1 \wedge \ldots \wedge \neg B_m]\!])\,] \\
\mathsf{pbc}_{EL}(GC; Com) & = & \mathsf{pbc}_{EL}(GC)\,;\,\mathsf{pbc}_{EL}(Com).
\end{array}$$

In the fourth case, it is assumed that each B_j denotes the guard of GC_j ($1 \le j \le m$). Note that it is safe to use there a generalised loop operator without explicit initialisation, since choice alternatives as well as inner loops are always guarded by their initial Boolean expression. Moreover, since parallel composition is also guarded (i.e., no loop body can immediately start with a parallel composition), there is no potential ‘nonsafeness’ problem, i.e., we may safely (in two senses of the word) use the operator box $\Omega^2_{\langle *]}$, which is defined as the symmetrical counterpart of $\Omega^2_{[* \rangle}$, shown in figure 35.

6.2.3 Actions and guarded commands

We still need to define the translations $\mathsf{pbc}_{EL}([\![BExpr]\!])$, $\mathsf{pbc}_{EL}([\![BExpr\,; x{:=}Expr]\!])$, and $\mathsf{pbc}_{EL}([\![x{:=}Expr]\!])$, to complete the definitions of $\mathsf{pbc}_{EL}(GC)$ and $\mathsf{pbc}_{EL}(Act)$. All definitions follow the same pattern. Instead of giving them in general, we only show some examples and refer the reader to [9] for the general case. Assume that variables x and y are declared by **var** $x, y : \{0, 1\}$; then we have the following translations. As the reader may recall, x_{kl} is supposed to denote the change of the value of x from k to l. Hence, the following should be self-explanatory:

$$\begin{array}{lcl} \mathsf{pbc}_{EL}([\![y{=}0]\!]) & = & (\,\{y_{00}\}\,) \\ \mathsf{pbc}_{EL}([\![y{=}0; x{:=}1{-}y]\!]) & = & (\,\{x_{01}, y_{00}\} \,\square\, \{x_{11}, y_{00}\}\,) \\ \mathsf{pbc}_{EL}([\![x{:=}1{-}y]\!]) & = & (\,\{x_{01}, y_{00}\} \,\square\, \{x_{11}, y_{00}\} \,\square\, \{x_{00}, y_{11}\} \,\square\, \{x_{10}, y_{11}\}\,). \end{array}$$

From the general theory presented in the previous sections, it follows that, for any *EL* program fragment P, $\mathsf{pbc}_{EL}(P)$ is a PBC expression and $\mathsf{box}_{\mathrm{PBC}}(\mathsf{pbc}_{EL}(P))$ is a static box (possibly up to ex-directedness, which may be transgressed in innocuous cases). Moreover, if P is a properly formed program (i.e., all variables are declared in outer blocks), then all transitions of $\mathsf{box}_{\mathrm{PBC}}(\mathsf{pbc}_{EL}(P))$ have label $\emptyset$.

In [9], it is shown how a more powerful concurrent language can be treated in a similar way, and also how high-level Petri nets [27] (rather than place/transition-nets) can be used as the target domain of the semantic function; however, this is beyond the scope of the present paper.

6.3 Proofs of distributed algorithms

The main purpose of a concurrent programming language such as *EL* is to express parallel programs and distributed algorithms. It is still acknowledged to be a difficult problem, in general, to state and prove the correctness of such programs and/or algorithms formally. An important goal of compositional translations such as that defined by the mapping $\mathsf{box}_{\mathrm{PBC}}(\mathsf{pbc}_{EL}(.))$ is to create a usable framework for making Petri net specific methods readily available – perhaps in addition to other methods – in order to solve this problem.

We may contrast expressing a distributed algorithm first in a programming language and then, by a standard translation, as a net, with an approach (exemplified by [41]) of 'massaging' an algorithm until it can be expressed as succinctly as possible by a net, independently of how it was expressed in the first place. Both approaches allow net theoretic means to be applied in proofs of correctness, and they both have their advantages and disadvantages. For instance, the second one often creates nets which are very well adapted to the algorithm, while the succinctness of the nets created by the first one is occasionally limited by the programming language. On the other hand, the first approach can be automatised more easily and allows the application of complementary methods such as the Owicki-Gries method [36] (which is known to be relatively complete) in addition to net theoretical methods [3].

In the remainder of this section we discuss briefly some (automatised) net-based correctness proofs of three different mutual exclusion algorithms: Peterson's algorithm [38], Dekker's algorithm [17] and Morris's algorithm [33].

6.3.1 Peterson's mutual exclusion algorithm

Figure 42 shows two parallel *EL* processes implementing Peterson's mutual exclusion protocol.[15] It is claimed that it is not possible for both processes to execute their respective critical sections concurrently; other claims are that there are neither deadlocks nor starvation, in the sense that if one of the processes has succeeded in executing a_1 or b_1, the other cannot prevent it from entering its critical section.

var $in_1, in_2\colon \{0,1\}$ (**init** 0); $hold\colon \{1,2\}$;

do	$[\![\mathbf{true}; in_1 := 1]\!]$;	a_1	‖	**do**	$[\![\mathbf{true}; in_2 := 1]\!]$;	b_1
	$[\![hold := 1]\!]$;	a_2	‖		$[\![hold := 2]\!]$;	b_2
	if $[\![in_2{=}0]\!] \,\Box\, [\![(hold \neq 1)]\!]$ **fi**;	a_3	‖		**if** $[\![in_1{=}0]\!] \,\Box\, [\![(hold \neq 2)]\!]$ **fi**;	b_3
	$CritSct_1$;		‖		$CritSct_2$;	
	$[\![in_1 := 0]\!]$	a_4	‖		$[\![in_2 := 0]\!]$	b_4
od			‖	**od**		

Figure 42: Peterson's algorithm with two processes.

The translation of this *EL* program (using the initial value $hold = 1$) into a box, as defined in section 6.2, is exemplified – in slightly simplified form – in figure 43. For instance, transition u_1 corresponds to a_1, transitions u_2 and u_3 correspond to a_2, transitions u_4 and u_5 correspond to a_3, and transition u_6 corresponds to a_4. Transitions v_j correspond to b_i in a similar way.

A Petri net based analysis of this algorithm employs four basic lemmata of the theory of Petri nets, namely: *(1) The number of tokens on an S-invariant is constant over transition occurrences*; *(2) A trap with at least one token can never be completely emptied of tokens*; *(3) The occurrence vector of a sequence reproducing a marking is a semi-positive T-invariant, and conversely, if a sequence is such that its occurrence vector is a T-invariant, then it reproduces the marking*; and *(4) In a bounded finite net, if there is some infinite occurrence, then there is an integer-valued semipositive T-invariant whose support (i.e., the transitions it assigns a nonzero number) equals the set of transitions occurring infinitely often in that occurrence*. These lemmata can be found in, e.g., [14].

The critical section property is satisfied. A detailed 'by-hand' verification of this fact is contained in [3]. We outline it here as follows. The net shown in Figure 43 is decomposable into five S-components, two for the sequential components and three for the variables *hold*, in_1 and in_2. This automatically yields five S-invariants which are initially marked with one token each. The net has more S-invariants than those, however; e.g., $\{in_1{=}0, p_2, p_3, p_4\}$ and $\{in_2{=}0, q_2, q_3, q_4\}$. The places p_4 and q_4 represent the critical sections. To prove the property of mutual exclusion, it has to be shown that $M(p_4){=}0 \vee M(q_4){=}0$ for every reachable marking M. The S-invariants of the net alone lack the power to prove this property, because if the twelve side condition arrows in figure 43 are omitted, then the set of S-invariants remains the same, but mutual exclusion is violated in the reduced net (for instance, by the sequence $u_1u_3u_4v_1v_3v_4$). Relationships between the two components of the program may enter the proof by the

[15] Variable initialisations are not in the syntax but can be accommodated easily.

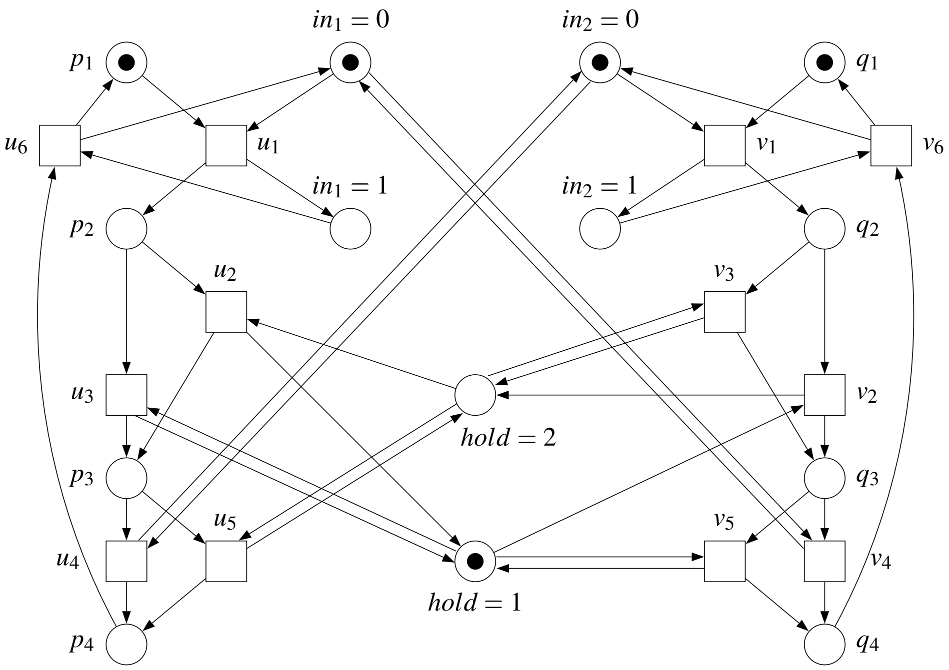


Figure 43: Translation of Peterson's algorithm into a net

trap method which was first described in [2] (see also [3, 7, 12, 41]). Two relevant traps are $\{in_1{=}0, p_2, hold{=}1, q_3\}$ and $\{in_2{=}0, q_2, hold{=}2, p_3\}$. Both carry at least one token initially, and by the second property cited above, none of them can be emptied of tokens.

Suppose now that some marking M is reachable from the initial marking M_0 such that a token is on p_4 and another token is on q_4. Using the first property mentioned above and the above set of S-invariants, it follows that $M(in_1{=}0) = M(p_3) = M(p_2) = 0$ and $M(in_2{=}0) = M(q_3) = M(q_2) = 0$. It then follows that at least one of the two traps is unmarked at M, which yields a contradiction.

An easy proof of deadlock-freeness is left to the reader.

The program is conditionally starvation-free. In order to demonstrate this, it needs to be shown that there is no infinite execution which, from a certain point on, continues to have a token on place p_2:

$$\sigma \;=\; M_0 t_1 M_1 t_2 M_2 \ldots \underbrace{M_{j-1} t_j M_j t_{j+1} \quad \ldots}_{1 = M_{j-1}(p_2) = M_j(p_2) = \ \ldots}$$

We may do this net-theoretically by using a method which was introduced in [12]. It consists of enumerating all T-invariants and showing, from the existence of the above infinite execution and the third and fourth properties, the existence of a certain T-invariant not in this set, which leads to a contradiction; the full argument is given in [3].

Automatic verification of the mutual exclusion property. The program shown in figure 43 can be fed (after a syntactic translation into the tool's input language) into the PEP partial order based model-checker [4, 37]. This model-checker translates any such program automatically into a net, calculates the McMillan finite prefix of that net [30] and executes Esparza's model-checker [19] with a given input formula of temporal logic, referring to the net (and hence, since the translation is tight, to the program as well). In this case, the formula to be checked is $\neg(\Diamond(p_4 \wedge q_4))$. On a SUN Ultra-1/140 with normal load, the execution times shown in table 4 were measured. Note that for any given program and net, the prefix has to be constructed only once, so that only the third line is relevant for any given formula (except the first one to be checked).

Task	Peterson (2 processes)	Dekker (2)	Morris (3)
Calculating the net	0.07 seconds	0.05 seconds	0.04 seconds
Building the finite prefix	0.01 seconds	0.05 seconds	1.19 seconds
Checking the formula	0.02 seconds	0.01 seconds	0.12 seconds

Table 4: Execution times for automatic verification by PEP [37].

6.3.2 Dekker's and Morris's mutual exclusion algorithms

Figure 44 shows Dekker's algorithm for mutual exclusion; the component c_2 arises from c_1 by exchanging all 1's with 2's (also in the indices). Morris's algorithm is shown in figure 45. Table 4 shows the execution times of automatic Petri net based verification of the mutual exclusion property for Dekker's algorithm with two processes and Morris's algorithm with three processes (and a correspondingly adapted temporal logic formula). 'By-hand' verification of Dekker's and Morris's algorithms using Petri net methods can be found in [12] and [44], respectively. The reader is also referred to the article by W. Reisig in this volume.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{var}\ in_1, in_2 : \{\mathbf{true}, \mathbf{false}\}\ (\mathbf{init\ false});\ hold : \{1,2\}\,;} \\
c_1 : & \mathbf{do} & [\![\mathbf{true}]\!]\,; [\![in_1 := \mathbf{true}]\!]; \\
& & \mathbf{do}\,[\![in_2]\!]\,; \quad \mathbf{if}\ [\![hold = 2]\!]\,; \quad [\![in_1 := \mathbf{false}]\!]; \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\,[\![hold = 1]\!]\ \mathbf{fi}; \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad [\![in_1 := \mathbf{true}]\!] \\
& & \qquad\qquad\ \ \Box\ [\![hold \neq 2]\!] \\
& & \qquad\qquad\ \mathbf{fi} \\
& & \mathbf{od}; \\
& & CritSct_1; \\
& & [\![hold := 2]\!]; \\
& & [\![in_1 := \mathbf{false}]\!] \\
& \mathbf{od} &
\end{array}
$$

Figure 44: Dekker's algorithm $c_{dekker} = c_1 || c_2$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{var}\ a, b : \{0,1\}\ (\mathbf{init}\ 1);\ m : \{0,1\}\ (\mathbf{init}\ 0);\ na, nm : \mathbf{N}\ (\mathbf{init}\ 0)\,;} \\
c_i : & P(b);\ [\![na := na + 1]\!];\ V(b); \\
& P(a);\ [\![nm := nm + 1]\!]; \\
& \ \ P(b);\ [\![na := na - 1]\!]; \\
& \ \ \mathbf{if}\ [\![na = 0]\!]; V(b); V(m)\ \Box\ [\![na \neq 0]\!]; V(b); V(a)\ \mathbf{fi}; \\
& P(m);\ [\![nm := nm - 1]\!]; \\
& CritSct; \\
& \mathbf{if}\ [\![nm = 0]\!]; V(a)\ \Box\ [\![nm \neq 0]\!]; V(m)\ \mathbf{fi}
\end{array}
$$

Figure 45: Morris's algorithm $c_{morris} = c_1 || \ldots || c_n$.

Acknowledgments

We thank Grzegorz Rozenberg for his patience and support, and Petra Gerdes for her assistance in drawing the figures.

References

[1] J. Baeten, W.P. Weijland: *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18 (1990).

[2] E. Best: Representing a Program Invariant as a Linear Invariant in a Petri Net. *Bulletin of the European Association of Theoretical Computer Science* Vol. 17, 2–11 (1982).

[3] E. Best: *Semantics of Sequential and Parallel Programs*. Prentice Hall (1996).

[4] E. Best: Partial Order Verification with PEP. Proc. *POMIV'96, Partial Order Methods in Verification*, G. Holzmann, D. Peled, V. Pratt (eds). American Mathematical Society, Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, 305–328 (1996).

[5] E. Best, R. Devillers: Sequential and Concurrent Behaviour in Petri Net Theory. Theoretical Computer Science Vol. 55/1, 87-136 (1988).

[6] E. Best, R. Devillers, J. Esparza: General Refinement and Recursion Operators for the Petri Box Calculus. Springer-Verlag, Lecture Notes in Computer Science Vol. 665, 130-140 (1993).

[7] E. Best, R. Devillers, J.G. Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. Advances in Petri Nets 1992, G.Rozenberg (ed.), Springer-Verlag, Lecture Notes in Computer Science Vol. 609, 21-69 (1992).

[8] E. Best, R. Devillers, M. Koutny: *Petri Net Algebra* (Working Title). Manuscript (1997).

[9] E. Best, H. Fleischhack, W. Frączak, R.P. Hopkins, H. Klaudel, E. Pelz: An M-net Semantics of $B(PN)^2$. Proc. *STRICT'95*, Berlin, J. Desel (ed.). Springer-Verlag, Workshops in Computing, 85–100 (1995).

[10] E. Best, M. Koutny: Solving Recursive Net Equations. Proc. ICALP-95, Springer-Verlag, Lecture Notes in Computer Science Vol.944, 605-623 (1995).

[11] G. Boudol, I. Castellani: Flow Models of Distributed Computations: Three Equivalent Semantics of CCS. *Information and Computation* Vol. 114(2), 247–314 (1994).

[12] G. Bruns, J. Esparza: Trapping Mutual Exclusion in the Box Calculus. *Theoretical Computer Science* Vol. 153/1–2, 95–128 (1995).

[13] P. Degano, R. De Nicola, U. Montanari: A Distributed Operational Semantics for CCS Based on C/E Systems. *Acta Informatica* Vol. 26, 59–91 (1988).

[14] J. Desel, J. Esparza: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science 40, Cambrige University Press (1995).

[15] R. Devillers: Construction of S-invariants and S-components for Refined Petri Boxes. Proceedings of Petri Nets'93, Chicago, LNCS 691, Springer-Verlag (1993).

[16] R. Devillers: S-invariant Analysis of Recursive Petri Boxes. *Acta Informatica* Vol. 32, 313-345 (1995).

[17] **Cited in:** E.W. Dijkstra: Cooperating Sequential Processes. *Programming Languages*, F. Genuys (ed.), 43–112. Academic Press (1968).

[18] E.W. Dijkstra: *A Discipline of Programming.* Prentice Hall (1976).

[19] J. Esparza: Model Checking based on Branching Processes. Habilitation, Hildesheim (1993). Published as: Model Checking Using Net Unfoldings. Proc. *TAPSOFT'93* (1993), M.C. Gaudel, J.P. Jouannaud (eds). Springer-Verlag, LNCS Vol. 668, 613–628 (1993). Full version in *Science of Computer Programming* Vol. 23, 151–195 (1994).

[20] H.J. Genrich, K. Lautenbach, P.S. Thiagarajan: Elements of General Net Theory. *Net Theory and Applications*, Proc. of the Advanced Course on General Net Theory of Processes and Systems, W. Brauer (ed.). Lecture Notes in Computer Science Vol. 84, Springer-Verlag, 21–163 (1980).

[21] U. Goltz: On Representing CCS Programs by Finite Petri Nets. Proc. MFCS'88, Springer-Verlag, Lecture Notes in Computer Science Vol. 324, 339-350 (1988).

[22] U. Goltz, R. Loogen: A Non-interleaving Semantic Model for Nondeterministic Concurrent Processes. *Fundamenta Informaticae* Vol. 14/1, 39–73 (1991).

[23] U. Goltz, W. Reisig: The Non-sequential Behaviour of Petri Nets. *Information and Control* Vol. 57/2–3, 125–147 (1983).

[24] J. Grabowski: On Partial Languages. *Fundamenta Informaticae* Vol. IV/2, 427–498 (1981).

[25] C.A.R. Hoare: *Communicating Sequential Processes*. Prentice Hall (1985).

[26] R. Janicki, P.E. Lauer: *Specification and Analysis of Concurrent Systems - the COSY Approach*. Springer-Verlag, EATCS Monographs on Theoretical Computer Science (1992).

[27] K. Jensen: Coloured Petri Nets. Basic Concepts. EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1992).

[28] M. Koutny, and E. Best: Operational Semantics for the Box Algebra. Hildesheimer Informatikbericht Nr.33/95 (October 1995). To appear in *Theoretical Computer Science* (1998).

[29] A. Mazurkiewicz: Trace Theory. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Advances in Petri Nets 1986, Part II, W. Brauer, W. Reisig, G. Rozenberg (eds.), Springer-Verlag, Lecture Notes in Computer Science Vol. 255, 279-324 (1987).

[30] K.L. McMillan: Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. Proc. *4th Workshop on Computer Aided Verification*, 164–174 (1992).

[31] R. Milner: *A Calculus of Communicating Systems*. Springer-Verlag, Lecture Notes in Computer Science Vol. 92 (1980).

[32] R. Milner: *Communication and Concurrency*. Prentice Hall (1989).

[33] J.H. Morris: A Starvation-free Solution to the Mutual Exclusion Problem. *Information Processing Letters* Vol. 8/2, 76–80 (1979).

[34] T. Murata: Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* Vol. 77/4, 541–580 (1989).

[35] E.-R. Olderog: *Nets, Terms and Formulas*. Cambridge Tracts in Theoretical Computer Science 23 (1991).

[36] S.S. Owicki, D. Gries: An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica* Vol. 6, 319–340 (1976).

[37] *PEP*. The home page of PEP (a Programming Environment Based of Petri Nets) is `http://www.informatik.uni-hildesheim.de/ pep/HomePage.html`.

[38] G.L. Peterson: Myths about the Mutual Exclusion Problem. *Information Processing Letters* Vol. 12/3, 115–116 (1981).

[39] G. Plotkin: A Structural Approach to Operational Semantics. DAIMI Technical Report FN-19, Computer Science Department, University of Århus (1981).

[40] W. Reisig: *Petri Nets. An Introduction*. EATCS Monographs on Theoretical Computer Science Vol. 3, Springer-Verlag (1985).

[41] W. Reisig: Modelling and Verification of Distributed Algorithms. Proc. *CONCUR'96*, U. Montanari, V. Sassone (eds), Springer-Verlag, LNCS Vol. 1119, 579–595 (1996).

[42] P.H. Starke: Processes in Petri Nets. *Elektronische Informationsverarbeitung und Kybernetik* Vol. 17/8–9, 389–416 (1981).

[43] D. Taubner: *Finite Representation of CCS and TCSP Programs by Automata and Petri Nets*. Springer-Verlag, Lecture Notes in Computer Science Vol. 369 (1989).

[44] B. Teßmer: *S- und T-Invarianten zum Nachweis von Eigenschaften paralleler Algorithmen*. Diplomarbeit (Januar 1996).

[45] W. Vogler: Partial Words versus Processes: a Short Comparison. *Advances in Petri Nets 1992*, G.Rozenberg (ed.). Springer-Verlag, Lecture Notes in Computer Science Vol. 609, 292–303 (1992).