Theoretical Computer Science 77 (1990) 97-129
North-Holland

TRANSFORMATIONS OF SEQUENTIAL SPECIFICATIONS INTO CONCURRENT SPECIFICATIONS BY SYNCHRONIZATION GUARDS

Ryszard JANICKI

Department of Computer Science and Systems, McMaster University, 1280 Main Street West, Hamilton, L8S 4K1 Ontario, Canada

Tomasz MÜLDNER

Jodrey School of Computer Science, Acadia University, Wolfville, B0P 1X0 Nova Scotia, Canada

Abstract. A transformation $\mathbb{C}$ of sequential specifications into concurrent specifications is defined. The sequential specification is in the form of a regular expression extended with a declaration of the actions that are independent and have the potential for concurrent execution. The concurrent specification is in the form of a product of regular expressions. It is proved that a concurrent specification resulting from the application of the transformation $\mathbb{C}$ to a sequential specification modified by inserting special actions, called synchronization guards, is behaviorally equivalent to the original specification. The programming language representation of a sequential specification is exemplified in a Pascal-like language, Banach.

1. Introduction

With the development of multiprocessors, such as the SEQUENT computer [38], research on concurrent systems has expanded immensely. There have been many attempts to define models of concurrent systems. On a more practical side, a number of programming languages providing constructs to express concurrency have been designed and implemented. Unfortunately, concurrent systems are difficult to deal with. For example, an *analysis* of a concurrent system requires the consideration of events that occur at the same time and with independent speeds. Similar difficulties are also apparent in the process of design of concurrent programs. This process appears to be rather complicated with the currently available concurrent languages, such as those based on the concept of a *monitor*, Modula [45], Pascal-Plus [44], Mesa [29], and Concurrent Euclid [14], and those based on *message passing*, CSP [13], CSP/80 [27], occam [15], Planet [8] and Joyce [6]. The programmer has to specify which actions are to be executed concurrently and then how those actions are to be synchronized. The latter specification is particularly cumbersome and error-prone.

In many cases, to implement a concurrent system S, we extract the sequential components of S, implement these, and then combine the sequential components into a complete implementation of S. Such a methodology is convenient for systems with inherent concurrent activities, for example operating systems or real-time

systems. However, there is large class of applications that are usually specified by sequential algorithms; for example, sorting and linear algebra algorithms, searching routines. For those applications, concurrency is often employed to improve performance but is not a primary goal. An alternative methodology is to describe a sequential system and then automatically develop a functionally equivalent concurrent system. The difference between these two methodologies is clear; in the former methodology, the user specifies synchronization details explicitly, while in the latter methodology the user may not be aware of the transformation from the sequential specification to the concurrent specification.

While we realize that a sequential algorithm does not always lead to an efficient concurrent algorithm, we demonstrate in this paper that, for many algorithms, our approach provides an effective way of obtaining a significant speed-up of the original specification. The research in this direction was originated in [16–19] which describe transformations of sequential systems into concurrent systems. Similar approach, but different techniques were proposed in [32, 33]. Two components of the description of a sequential system are required, namely a specification of the sequential behavior and a declaration of actions that are independent and so have the potential for concurrent execution. Such a specification is *potentially concurrent* and so in this paper is called a *p-concurrent* specification. A p-concurrent specification is called *proper* if it is functionally equivalent to a corresponding concurrent specification. As of now, we do not know general sufficient and necessary conditions for a p-concurrent specification to be proper; in [19] such conditions are given for the special subclass of p-concurrent specifications (in the form of COSY systems).

In order to explain this approach, we now recall from [16, 35] the example of a factorial. Let us consider the sequential, iterative factorial scheme to compute $x!$: $z := 1$; while $(x > 0)$ do begin $z := z * x$; $x := x - 1$ end. Since the body of this loop consists of two *dependent* actions, it is modified to the form in which independent actions can be identified: $z := 1$; while $(x > 0)$ do begin $y := x$; $z := z * y$; $x := x - 1$ end. Now, this program is translated to the following regular expression (see Section 2): $(z := 1); ((x > 0); (y := x); (z := z * y); (x := x - 1))^*; (x \leqslant 0)$. The pairs of actions that are mutually independent are $((z := 1),(x > 0))$, $((z := 1),(y := x))$, $((z := 1), (x := x - 1))$, $((z := 1),(x \leqslant 0))$, $((z := z * y),(x := x - 1))$, $((z := z * y),(x \leqslant 0))$, $((x > 0),(z := z * y))$. From these pairs one can derive the three maximal sets of mutally dependent actions: $\{(x > 0), (y := x), (x := x - 1), (x \leqslant 0)\}$, $\{(z := 1), (z := z * y)\}$, $\{(y := x), (z := z * y)\}$. Now, the three sequential components of the concurrent program being constructed can be obtained by erasing actions that do not appear in the above sets. Thus, we obtain a program$(((x > 0), (y := x); (x := x - 1))^*; (x \leqslant 0)(\|)(z := 1); (z := z * y)^*(\|)((y := x); (z := z * y))^*)$ and it is easy to prove that this program really implements a parallel factorial (see [35]). Note, that the three components of the concurrent program synchronize through the so-called hand-shake synchronization, that is if the same action, such as $y := x$, occurs in more than one component, it has to be executed *at the same time* in all such components; for details see Definition 2.5 and Example 2.6.

In some cases the above transformation results in a concurrent specification that is *not* proper. In this paper we extend the ideas described above and show that by introducing special actions, called *synchronization guards*, every p-concurrent specification can be transformed into an equivalent concurrent specification. The synchronization actions prevent some paths in the concurrent execution from performing. (Note that synchronization guards have also proved to be useful for investigating parallel devices recognizing trace languages [24].) We also show that a sequential specification can be represented in the *sequential* Pascal-like programming language, called Banach. This language has been designed in such a way that the programmer does not have to be concerned about the synchronization details [21, 22]. In Banach, the basic unit of specification is called an event. Events resemble Ada's packages [40] or Simula's classes [3], in that they have parameters, local data structures and bodies. Events can be initiated and they can participate in standard sequential operations, such as composition, conditional choice, or iteration. A Banach program consists of two parts; a description of the behavior of events and a declaration of the independent events. Independent events must not engage in concurrent actions that could result in ill-defined results. For example, two events, one of which reads the value of a variable x, and the other which updates the value of x should not be declared as independent.

To illustrate the main idea of our project, let us describe its four components. At the linguistic level, there are two components (see Fig. 1). In our theory, Banach specifications are modelled by p-concurrent regular expressions, and concurrent programs are modelled by concurrent regular expressions. Thus, at the theory level we have the following components shown in Fig. 2. The main result of this paper shows that the transformation $\mathbb{C}$ yields an equivalent concurrent expression, which gives theoretical background for the implementation of Banach.

Until recently, the problem of deriving an equivalent concurrent system from the sequential specification has not been very popular among the computer community, and this still seems to be the case among theoreticians. However, there have been many recent research projects on this subject (see [1, 4, 8, 9, 41] and others). To explain the reason for the interest in this kind of research, we quote [12]: "... there is an economic fact of life that cannot be ignored, namely that large companies

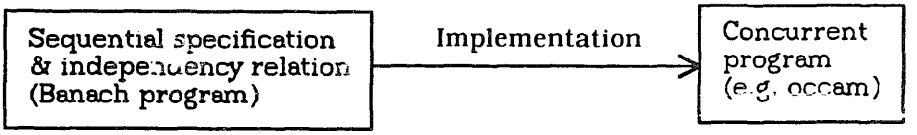

Fig. 1.

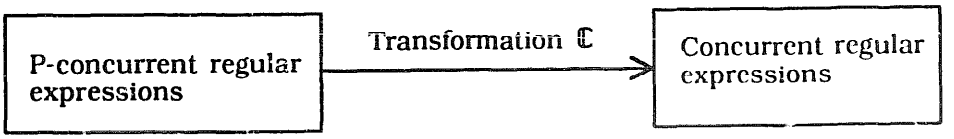

Fig. 2.

with many hundreds of man-years invested in sequential software will not invest substantially in parallel hardware until there is a convincing 'migration route' to such machines...". Unfortunately, most results, such as those in [4, 9, 12, 41] are concerned with a very specific hardware or software, and only a few, such as [1], try to find solutions that are as general as ours.

This paper is organized as follows. In Section 2 we present a description of the formalism used in this paper: regular expressions; concurrent regular expressions; and p-concurrent regular expressions. We also define a transformation $\mathbb{C}$ of p-concurrent regular expressions into concurrent regular expressions. In Section 3 we define synchronized p-concurrent regular expressions and a transformation Π of p-concurrent regular expressions into synchronized p-concurrent regular expressions. We prove the main result of our paper which states that for every p-concurrent regular expression PCR, the synchronized concurrent regular expression Π(PCR) is proper, that is, equivalent to $\mathbb{C}(\Pi(\text{PCR}))$. In that section we also show an application of our technique to the well-known dining philosophers problem. Section 4 describes the Banach programming language. Examples provided in this section explain the main ideas behind the design of Banach. In Section 5 we show modelling of Banach programs by p-concurrent regular expressions. Some results of this paper have been announced in [20–23, 37].

2. Regular and concurrent regular expressions

Let us first introduce some notations used in this paper. If A is an arbitrary alphabet, the elements of A are denoted by a,b,c (with indices, if necessary.) By A^* we denote the set of all strings over A, including the empty string ε, and the elements of A^* are denoted by x,y,z (with indices, if necessary.) By A^+, we denote $A^* - \{\varepsilon\}$, and by $\#(_a x)$ we denote the number of occurrences of the symbol a in the string x. For any language $L \subseteq A^*$, let $\text{Pref}(L) = \{x \in A^*: (\exists y \in A^*)(xy \in L)\}$. For every regular expression R, let $\mathcal{L}(R)$ denote the language defined by R. Finally, for a regular expression R (also, for a string, a language or any other mathematical construct), by A_R we denote the alphabet of R.

Definition 2.1. A pure *sequential specification* is a regular expression R over a finite alphabet A_R with the three operators ";", "," and "*". The elements of A_R are *atomic* actions, or events. In this paper, we use ";" to denote the concatenation, "," to denote the choice, and "*" to denote the iteration. Unlike the traditional notation we assume that "," has higher precedence than ";". Our approach is based on the COSY model (see e.g. [26, 30, 31], or [2]) in which the same convention has been used. Thus, our "$a,b;c$" is the same as the traditional expression "$(a \cup b)c$".

Clearly, well-known conditional and iterative constructs can be expressed in our notation (see [11, 34]): if α then I else J as $(\alpha;I)$, $(\underline{\alpha};J)$, and while α do I as $(\alpha;I)^*$, $\underline{\alpha}$ where $\underline{\alpha}$ denotes the negation of α.

In this paper, we define the semantics of sequential and concurrent specifications in terms of two components which are, respectively, intermediate results or *histories*, and final results or *resulting histories*. We require the following.

Requirement 2.2. Every initial segment, or prefix of history is history, and every resulting history is history.

Definition 2.3. The set of *resulting histories* of a pure sequential specification R, denoted by $\mathrm{RH}(R)$ is defined as the language $\mathscr{L}(R)$. The set of *histories* of R, denoted by $\mathrm{H}(R)$ is defined as $\mathrm{Pref}(\mathrm{RH}(R))$.

Clearly, the above definition satisfies Requirement 2.2. Now, we define a concurrent regular expression.

Definition 2.4. A *concurrent regular expression* is of the form CR: $R_1 \| \cdots \| R_n$ where R_i for $i = 1, \ldots, n$, are regular expressions. We often write A_i instead of A_{R_i}. Let $A_{\mathrm{CR}} = \bigcup_i A_i$ denote the alphabet of CR.

Since in this paper concurrent specifications are often expressed by concurrent regular expressions, unless specified otherwise we shall identify these two concepts.

The semantics of concurrent expressions can be defined using vector sequences [42]. The following notations are useful.

Definition 2.5. Let CR: $R_1 \| \cdots \| R_n$, $A = A_{\mathrm{CR}}$. We define a concatenation of two vectors componentwise:

$$(x_1, x_2, \ldots, x_n)(y_1, y_2, \ldots, y_n) = (x_1 y_1, x_2 y_2, \ldots, x_n y_n),$$

where

$$(x_1, x_2, \ldots, x_n), (y_1, y_2, \ldots, y_n) \in A_1^* \times A_2^* \times \cdots \times A_n^*.$$

Let $h_i : A^* \to A^*$ be a projection homomorphism defined by an extension of

$$h_i(a) = \begin{cases} a & \text{if } a \in A_i, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Now, let $\_ : A^* \to A_1^* \times A_2^* \times \cdots \times A_n^*$ be the mapping defined by $\underline{x} = [h_1(x), \ldots, h_n(x)]$ for $x \in A^*$. For any language $L \subseteq A^*$, let vect be the following mapping $\mathrm{vect}(L) = \{\underline{x} : x \in L\}$. The elements of $\mathrm{vect}(A)$ are called *vector events of CR*, the elements of $\mathrm{vect}(A^*)$ are called *strings of vector events of CR*. For every $V \subseteq \mathrm{vect}(A^*)$, let $\mathrm{Pref}(V) = \{\underline{x} \in \mathrm{vect}(A^*) : (\exists \underline{y} \in \mathrm{vect}(A^*))(\underline{xy} \in V)\}$.

Example 2.6. For CR_1: $a;b \| a;c$ we have $A_R = \{a,b,c\}$ and there are three vector events, $\mathrm{vect}(A) = \{[a,a], [b, \varepsilon], [\varepsilon,c]\}$. The string $\underline{abc}$ of vector events represents a possible concurrent execution of the actions b and c, $\underline{abc} = [ab, ac] = \underline{acb} \in \mathrm{vect}(A^*)$.

Definition 2.7. Let CR: $R_1 \| \cdots \| R_n$ be a concurrent regular expression. We define the *independency relation* of CR, $I_{CR} \subseteq A_{CR} \times A_{CR}$ as follows $(a, b) \in I_{CR}$ iff $(a \neq b)$ & $(\forall i)$ $(a \notin A_i \vee b \notin A_i)$.

We extend independency relation I_{CR} to the relation $\sim$ on A^*. This relation allows us to not distinguish between two sequences of actions if they differ only in the order of executions of independent actions (compare [35, 36]). For $i = 1, 2, \ldots, n$, $a_1 a_2 \ldots a_i a_{i+1} \ldots a_n \sim a_1 a_2 \ldots a_{i+1} a_i \ldots a_n$ if $(a_i, a_{i+1}) \in I_{CR}$. It turns out (see [26]) that for the reflexive and transitive closure $\approx$ of the relation $\sim$ ($\approx = \sim^*$) we have $x \approx y$ iff $\underline{x} = \underline{y}$. Thus, in the strings of vector events, we do not distinguish between histories which represent the same activity of a concurrent system. For Example 2.6, we have $abc \approx acb$.

Definition 2.8. Let CR: $R_1 \| \cdots \| R_n$ be a concurrent specification. The semantics of CR is defined by the set of *resulting histores*, $\mathrm{RH(CR)} = \mathrm{vect}(A^*) \cap \mathrm{RH}(R_1) \times \cdots \times \mathrm{RH}(R_n)$, and the set of *histories* $\mathrm{H(CR)} = \mathrm{vect}(A^*) \cap \mathrm{H}(R_1) \times \cdots \times \mathrm{H}(R_n)$.

It is easy to show that the above definition satisfies Requirement 2.2 (see also [19]). For Example 2.6 we have $\mathrm{RH(CR_1)} = \{\underline{abc}\}$, $\mathrm{H(CR_1)} = \{\underline{abc}, \underline{ab}, \underline{ac}, \underline{a}, \underline{\varepsilon}\}$.

Corollary 2.9 (see [26] for details)

(1) $\mathrm{RH(CR)} = \{\underline{x} \in \mathrm{vect}(A^*) : (\forall i) h_i(x) \in \mathrm{RH}(R_i)\}$,

(2) $\mathrm{H(CR)} = \{\underline{x} \in \mathrm{vect}(A^*) : (\forall i) h_i(x) \in \mathrm{H}(R_i)\}$.

Example 2.10. For $\mathrm{CR_2}$: $(a;e),(b;f) \| (c;e),(d;f)$ we have two resulting histories $\mathrm{RH(CR_2)} = \{\underline{ace}, \underline{bdf}\}$. There are two histories that lead to a *deadlock* (that is, the system comes to a halt before it reaches a final state), that is $\mathrm{H(CR_2)} = \mathrm{Pref}(\mathrm{RH(CR_2)} \cup \{\underline{ad}, \underline{bc}\})$.

The above semantics, described in terms of vector firing sequences, can be equivalently described using Mazurkiewicz's traces (see [35, 36]). Both models use partially commutative monoids introduced in [7] (see also [28]) to model partial orders. More details can be found in [24, 36, 43].

Our goal is to describe transformations of sequential specifications into concurrent specifications. As mentioned before, the required sequential specification consists of two components: a description of sequential behavior and a declaration of independent actions. The first component is in the form of a pure sequential specification, the second component identifies sets of dependent actions. In this paper, sequential specifications are expressed using potentially concurrent, or briefly p-concurrent regular expressions.

Definition 2.11. A *p-concurrent regular expression* PCR is a regular expression R over the alphabet which is partitioned into a finite number of not necessarily disjoint subsets. Thus, a p-concurrent regular expression is a pair $\mathrm{PCR} = [R, \{A_1, A_2, \ldots, A_n\}]$ where $A_R \subseteq A_1 \cup A_2 \cup \cdots \cup A_n$.

Note that the two components of a p-concurrent regular expression are not interrelated, that is the specification of the sequential behavior is independent from the choice of the sets A_i. Clearly, two actions are independent if they do not belong to the same set A_i.

Definition 2.12. Let $\mathrm{PCR} = [R, \{A_1, A_2, \ldots, A_n\}]$ be a p-concurrent expression. We define the *independency relation* of PCR, $I_{\mathrm{PCR}} \subseteq A_R \times A_R$ as $(a, b) \in I_{\mathrm{PCR}}$ iff $(a \neq b)\ \&\ (\forall i)\ (a \notin A_i \vee b \notin A_i)$.

Next we define the semantics of p-concurrent regular expressions in terms of histories and resulting histories. To explain the intuition of these definitions, let us recall the following example from [19]. For PCR_0: $[R_0: ((a;b;c)^*), \{A_1 = \{a,b\}, A_2 = \{a,c\}\}]$ one can define the set of resulting histories of R_0 as the language $\mathscr{L}(R_0)$: $\mathrm{RH}(R_0) = \{abc\}^* = \{\varepsilon, abc, abcabc \ldots\}$ and the set of histories of R_0 as the set of prefixes of the above set: $\mathrm{H}(R_0) = \{abc\}^*\{ab, a, \varepsilon\} = \{\varepsilon, a, ab, abc, abca, abcab \ldots\}$. We now wish to apply the operation vect to obtain strings of vector events. The set $\mathrm{vect}(\mathrm{RH}(R_0))$ satisfies Requirement 2.2, and so can be used to define the set of resulting histories. However, the set $\mathrm{vect}(\mathrm{H}(R_0))$ is not prefix-closed because $\underline{acb} \in \mathrm{vect}(\mathrm{H}(R_0))$ but $\underline{ac} \notin \mathrm{vect}(\mathrm{H}(R_0))$. The histories defined by the p-concurrent regular expression should "approximate" the set $\mathrm{vect}(\mathrm{H}(R_0))$, because the set $\mathrm{H}(R_0)$ defines the histories of R_0, and the operation "_", of taking vectors, defined by the distribution $\{A_1, A_2, \ldots, A_n\}$ is an operation which forgets about superfluous sequentialization. The best approximation of the set $\mathrm{vect}(\mathrm{H}(R_0))$, closed under Pref is the least set containing $\mathrm{vect}(\mathrm{H}(R_0))$ and, at the same time, closed under Pref, that is the set $\mathrm{Pref}(\mathrm{vect}(\mathrm{RH}(R_0)))$.

Definition 2.13. The set of *resulting histories* of the p-concurrent expression $\mathrm{PCR} = [R, \{A_1, A_2, \ldots, A_n\}]$ is defined as $\mathrm{RH}(\mathrm{PCR}) = \mathrm{vect}(\mathrm{RH}(R))$ and the set of *histories* of PCR is defined as $\mathrm{H}(\mathrm{PCR}) = \mathrm{Pref}(\mathrm{vect}(\mathrm{RH}(R))) = \mathrm{Pref}(\mathrm{RH}(\mathrm{PCR}))$.

Example 2.14. For PCR_1: $[(a;b), \{A_1 = \{a\}, A_2 = \{b\}\}]$, $\mathrm{RH}(\mathrm{PCR}_1) = \{\underline{ab}\}$, $\mathrm{H}(\mathrm{PCR}_1) = \{\underline{ab}, \underline{a}, \underline{b}, \underline{\varepsilon}\}$.

Now, we describe a transformation $\mathbb{C}$ of p-concurrent expressions into concurrent expressions (compare [19]).

Definition 2.15. Consider the p-concurrent expression $\mathrm{PCR} = [R, \{A_1, A_2, \ldots, A_n\}]$. Let $\mathbb{C}(\mathrm{PCR})$ be the set of concurrent expressions $R_1 \| \cdots \| R_n$ such that for $i = 1, 2, \ldots, n$, R_i satisfies the condition $\mathrm{RH}(R_i) = h_i(\mathrm{RH}(R))$.

There may be more than one element of $\mathbb{C}(\mathrm{PCR})$. To derive expressions R_i satisfying the above condition, one possible algorithm is to replace in R those actions that do not appear in A_i by ε and use equivalence rules for regular expressions, such as $\mathrm{expr}_1, \mathrm{expr}_2 \equiv \mathrm{expr}_2, \mathrm{expr}_1$ or $\mathrm{expr};\varepsilon \equiv \mathrm{expr} \equiv \varepsilon;\mathrm{expr}$. For

Example 2.14 there are the following elements of $\mathbb{C}$(PCR): $(a;\varepsilon \| b:\varepsilon)$, $(a \| b)$, $(a;\varepsilon \| b)$, $(a \| b;\varepsilon)$.

Definition 2.16. For every p-concurrent expression PCR, and $CR \in \mathbb{C}(PCR)$ let $RH(\mathbb{C}(PCR)) = RH(CR)$ and $H(\mathbb{C}(PCR)) = H(CR)$.

From the definition of $\mathbb{C}$ it follows that for any $\xi_1, \xi_2 \in \mathbb{C}(PCR)$ we have $RH(\xi_1) = RH(\xi_2)$ and $H(\xi_1) = H(\xi_2)$, so both $RH(\mathbb{C}(PCR))$ and $H(\mathbb{C}(PCR))$ are defined in an unambiguous way, i.e. are independent of a particular choice of $CR \in \mathbb{C}(PCR)$.

Definition 2.17. We say that a p-concurrent regular expression $PCR = [R, \{A_1, A_2, \ldots, A_n\}]$ is *proper* if $\mathbb{C}(PCR)$ is equivaler[t] to PCR, that is $H(PCR) = H(\mathbb{C}(PCR))$ and $RH(PCR) = RH(\mathbb{C}(PCR))$.

From the preceding discussion, it follows that the above two equalities are well-defined. Let us explain why *proper* p-concurrent regular expressions are so important. In our model, a regular expression R represents a pure sequential specification, and PCR: $[R, \{A_1, A_2, \ldots, A_n\}]$ represents a sequential specification. Since $\mathbb{C}$ preserves behavioral properties of *proper* expressions only, when PCR is proper, it can be implemented by any $CR \in \mathbb{C}(PCR)$.

Example 2.18. Let PCR_2: $[(a;b;c), \{A_1 = \{a,b\}, A_2 = \{a,c\}\}]$. Then $\mathbb{C}(PCR_2) \ni CR_{2'}$: $a;b \| a;c$ and it is easy to see that PCR_2 is proper. Note that $CR_{2'}$ is identical to CR_1 from Example 2.6.

Now we give examples of PCRs which are not proper. In general, a PCR and a resulting CR may have different sets of histories and identical sets of resulting histories, or identical sets of histories and different sets of resulting histories.

Example 2.19. Let PCR_3: $[((a;c;e),(b;d;f)), \{A_1 = \{a,e,b,f\}, A_2 = \{c,e,d,f\}\}]$. Then $\mathbb{C}(PCR_3) \ni CR_{3'}$: $(a;e),(b;f) \| (c;e),(d;f)$, for which the sets of resulting histories are identical, but the set of histories of $CR_{3'}$ includes the sequence ad (leading to a deadlock), which clearly is not a history of PCR_3 (note that $CR_{3'}$ is identical to CR_2 from Example 2.10). To explain in more detail the reason why PCR_3 is not proper we use Petri nets (for the definition see [39]) to represent our specifications. PCR_3 can be represented by the Petri net shown in Fig. 3. After erasing all the actions that do not belong to sets A_1 and A_2 we have two sequential nets (Fig. 4).

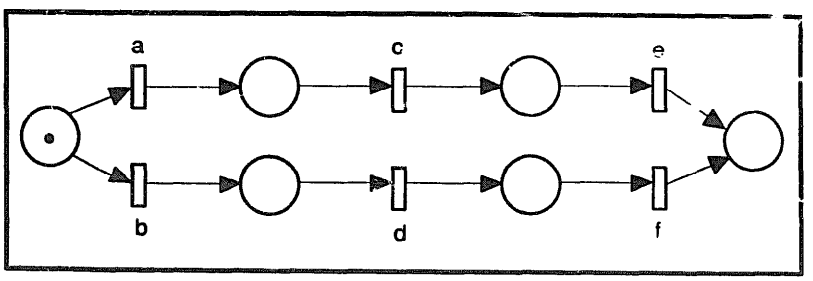

Fig. 3.

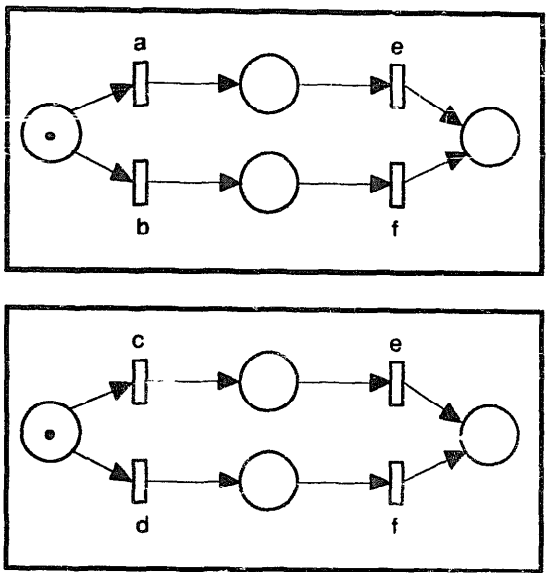

Fig. 4.

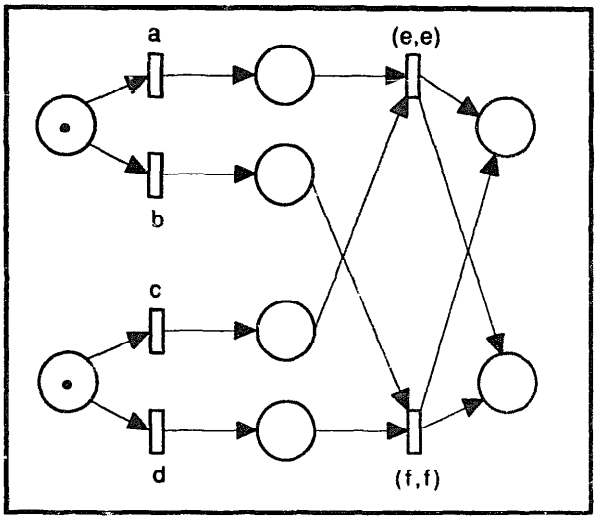

Fig. 5.

Now these two nets can be composed using the construction described in [25, p. 219] and shown in Fig. 5. It should be now clear why $\mathbb{C}(\mathrm{PCR}_3)$ is not proper: two independent actions occurring in branches of the alternative, namely a and d, are mapped by $\mathbb{C}$ to different components of the parallel construct $\|$ allowing *both* these actions to execute.

For the sake of completeness we recall from [19], necessary condtions for PCR to be proper.

Definition 2.20. Let $\mathrm{PCR}=[R,\{A_1,A_2,\ldots,A_n\}]$. Denote by conf (standing for conflict) the relation in $A_R \times A_R$ defined as follows:

$$(a,b)\in \mathrm{conf} \text{ iff } (\exists x\in A_{R^*})(\underline{xa}\in \mathrm{H(PCR)} \,\&\, \underline{xb}\in \mathrm{H(PCR)}$$

$$\&\, \underline{xab}\notin \mathrm{H(PCR)}) \,\&\, a\neq b).$$

If PCR is proper then $\text{conf} \cap I_{PCR} = \emptyset$. As the following example shows, this condition is necessary but *not* sufficient.

Example 2.21. For PCR_4: $[((a;b),(a;a;b;b)), \{A_1 = \{a\}, A_2 = \{b\}\}]$, the relation conf is empty, but PCR_4 is *not* proper.

Now we give an example of a PCR for which the sets of histories are identical, but resulting histories differ.

Example 2.22. For PCR_5: $[((a;b)^*), \{A_1 = \{a\}, A_2 = \{b\}\}]$ we have $\mathbb{C}(PCR_5) \ni CR_{5'}$: $a^* \| b^*$.

Resulting histories of PCR_5 must have the same number of occurrences of a and b, while resulting histories of $CR_{5'}$ contain an arbitrary number of these actions. The reason that the above PCR_5 is not proper is similar to that explained in Example 2.19: two independent actions were mapped by $\mathbb{C}$ to two parallel components. Thus, some actions from the nth step may be executed before all the actions from the previous step have been completed (see Fig. 6).

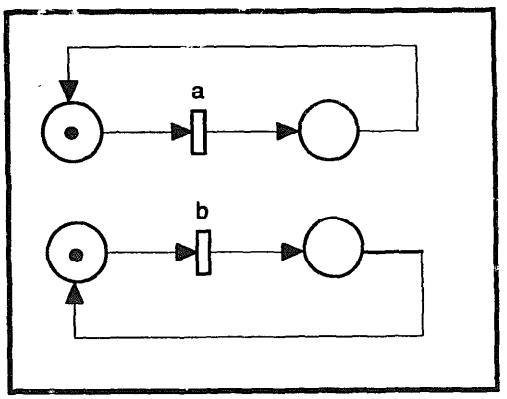

Fig. 6.

3. Synchronized p-concurrent regular expressions

To deal with the problems described at the end of the previous section, we introduce *synchronization guards* which are inserted into the original p-concurrent expressions (we assume that synchronization actions do not belong to the alphabet of PCR). These actions help to control the concurrent execution. Recall that in the above examples, p-concurrent regular expressions were *not* proper because of the confusion between the choice and the loop operators in the way in which $\mathbb{C}$ interprets

the independency relation. For example, independent actions occurring in branches of the choice were mapped by $\mathbb{C}$ to different components of the parallel construct $\|$. To recover this, synchronization guards are inserted into alternatives and loops and, by definition are in conflict with the bodies of these constructs. The synchronization guards will occur in two contexts (for a formal definition see Definition 3.4) as shown in Fig. 7:

- in alternatives, where they are inserted as the first action of one of the branches of these alternatives: branch_1, $(\Delta; \text{branch}_2)$
- in loops, they are inserted at the beginning of the body: $(\Delta; \text{body})^*$

Example 3.1. Consider PCR_5 from Example 2.22 with the additional synchronization guard Δ, PCR_6: $[((\Delta;a;b)^*), \{A_1=\{a,\Delta\}, A_2=\{b,\Delta\}\}]$. Note that Δ is in conflict with both a and b and so $\mathbb{C}(PCR_6) \ni CR_{6'}$: $(\Delta;a)^* \| (\Delta;b)^*$. It is easy to see that PCR_6 *is* proper (see Fig. 8).

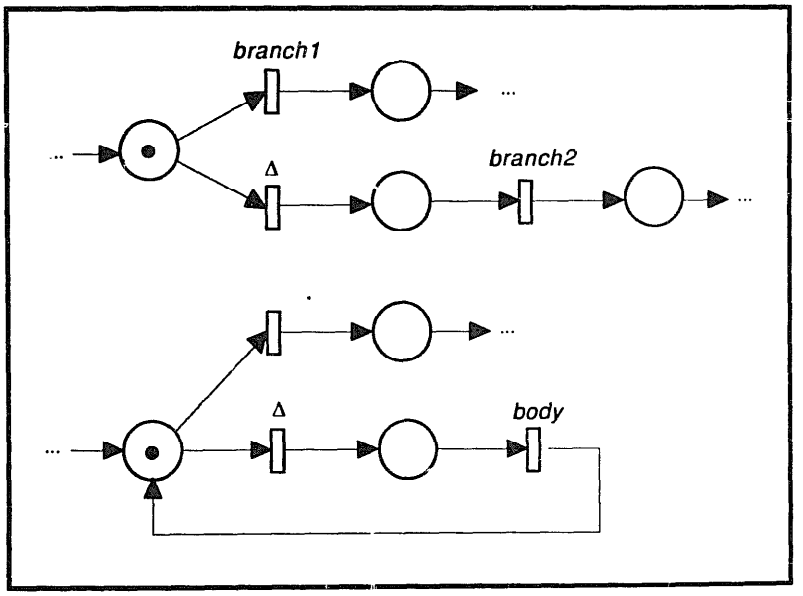

Fig. 7.

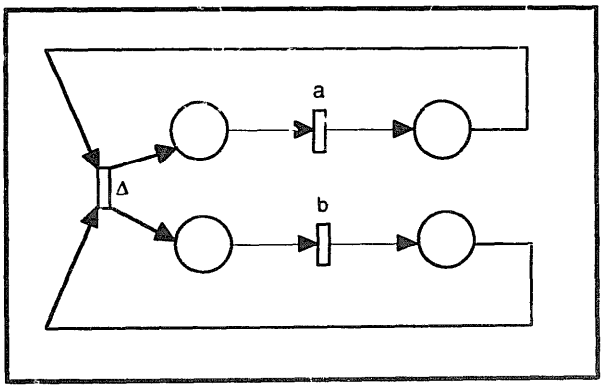

Fig. 8.

It turns out that in general some additional fine tuning is necessary.

Example 3.2. Let $PCR_7: [(a,b), \{A_1 = \{a\}, A_2 = \{b\}\}]$. Here, $\mathbb{C}(PCR_7) \ni CR_{7'}: a,\varepsilon \parallel b,\varepsilon$ and clearly PCR_7 is not proper. If we tried to deal with this problem as explained above, i.e. inserted the synchronization guard Δ in both the branches of the choice, we would obtain $PCR_{7'}: [(a,(\Delta;b)), \{A_1 = \{a,\Delta\}, A_2 = \{b,\Delta\}\}]$, which is proper. However, assuming that the comma operator is left-associative, i.e. (a,b,c) represents $((a,b),c))$, if we consider $PCR_8: [(a,b,c), \{A_1 = \{a\}, A_2 = \{b\}, A_3 = \{c\}\}]$ then we would obtain $PCR_{8'}: [(a,(\Delta;b),(\Delta;c)), \{A_1 = \{a,\Delta\}, A_2 = \{b,\Delta\}, A_3 = \{c,\Delta\}\}]$, which is not proper because $\mathbb{C}(PCR_{8'}) \ni CR_{8'}: a,\Delta,\Delta \parallel \varepsilon,(\Delta;b),\Delta \parallel \varepsilon,\Delta,(\Delta;c)$ and $(b,c) \in \text{conf} \cap I_{PCR}$.

Thus, we shall require that the synchronization guards are *unique*. The expression PCR_8 would now be transformed into the following proper expression:

$$PCR_9: [((a,(\Delta_1;b)),(\Delta_2;c)), \{A_1 = \{a,\Delta_1,\Delta_2\}, A_2 = \{b,\Delta_1,\Delta_2\}, A_3 = \{c,\Delta_1,\Delta_2\}\}].$$

In addition, it turns out that we must not assume that the synchronization guards are in conflict with *all* other actions which are not involved in loops and alternatives considered (thus, the above expression PCR_9 will be modified, see Example 3.10).

Example 3.3. Let $PCR_{10}: [(a;b,c), \{A_1 = \{a\}, A_2 = \{b,c\}\}]$. The concurrency in the expression $PCR_{10'}: [(a;b,(\Delta;c)), \{A_1 = \{a,\Delta\}, A_2 = \{b,c,\Delta\}\}]$ would be unnecessarily limited, because $\mathbb{C}(PCR_{10}) \ni CR_{10'}: a \parallel b,c$ and $\mathbb{C}(PCR_{10'}) \ni CR_{10''}: a;\varepsilon,\Delta \parallel b,(\Delta;c)$ and a and Δ cannot be performed simultaneously, i.e. $(a,\Delta) \notin H(CR_{10''})$.

Thus, we assume that the synchronizing guard is in conflict with other actions in the alternative, or the body of the loop, but not necessarily with the remaining actions in the entire expression. In Example 3.3, we assume that Δ is in conflict only with b and c, i.e. $A_1 = \{a\}$, $A_2 = \{b,\Delta,c\}$ and $\mathbb{C}(PCR_{10'}) \ni CR_{10''}: a \parallel b,(\Delta;c)$. Clearly, $(a,\Delta) \in H(CR_{10''})$.

Now we formulate the necessary formalism to present the ideas described above. We first define the class Ω of synchronized p-concurrent regular expressions.

Definition 3.4. Let $A_1, A_2, \ldots, A_n$ be alphabets, and let $i(a) = \{j: a \in A_j\}$. The class $\Sigma(A_1, A_2, \ldots, A_n)$ of synchronized regular expressions is defined by the following grammar:

$$\text{expr} ::= \text{el} \mid (\text{expr};\text{el})$$

$$\text{el} ::= a \mid (\Delta;\text{expr})^* \mid (\text{expr}, (\Delta;\text{expr}))$$

where a stands for any atomic action and the following conditions are satisfied:

- for each loop $(\Delta;\text{expr})^*$: $(\forall b \in A_{\text{expr}})(i(b) \subseteq i(\Delta))$;
- for each alternative $(\text{expr}_1,(\Delta;\text{expr}_2))$: $(\forall b \in A_{\text{expr}_1} \cup A_{\text{expr}_2})(i(b) \subseteq i(\Delta))$;
- synchronization guards are unique.

We write just Σ rather than $\Sigma(A_1, A_2, \ldots, A_n)$ if it does not lead to confusion.

Definition 3.5. A p-concurrent regular expression PCR: $[R, \{A_1, A_2, \ldots, A_n\}]$ is called a *synchronized p-concurrent regular expression*, PCR $\in \Omega$ if $R \in \Sigma(A_1, A_2, \ldots, A_n)$.

Theorem 3.6. *Every synchronized p-concurrent regular expression from the class Ω is proper.*

The proof is given in Appendix A.

Now we define a transformation Π from the set of all p-concurrent ε-free regular expressions into Ω. The mapping Π inserts synchronization guards as described above. Formally, this mapping is defined as follows.

Definition 3.7. Let PCR: $[R, \{A_1, A_2, \ldots, A_n\}]$ be a p-concurrent ε-free regular expression and let G be an infinite alphabet such that $G \cap A = \emptyset$, where $A = A_1 \cup A_2 \cup \cdots \cup A_n$. Elements of G are called synchronization guards and are denoted by Δ (with indices if necessary). First we define the mapping Φ from the set of regular expressions into Σ and the mappings $G_1, G_2, \ldots, G_n$ from the set of regular expressions into subsets of G. Mappings Φ and $G_1, G_2, \ldots, G_n$ are defined inductively:

(1) if R is an atomic action a, then $\Phi(a) = a$, $G_i(a) = \emptyset$, $i = 1, 2, \ldots, n$;

(2) $\Phi(R_1, R_2) = \Phi(R_1), (\Delta;\Phi(R_2))$,

$$G_i(R_1, R_2) = \begin{cases} G_i(R_1) \cup G_i(R_2) \cup \{\Delta\} & \text{if } (A_{R_1} \cup A_{R_2}) \cap A_i \neq \emptyset, \\ G_i(R_1) \cup G_i(R_2) & \text{otherwise,} \end{cases}$$

for $i = 1, 2, \ldots, n$, where Δ is any element of G such that $\Delta \notin A \cup G_i(R_1) \cup G_i(R_2)$;

(3) $\Phi(R_1;R_2) = \Phi(R_1);\Phi(R_2)$ and for $i = 1, 2, \ldots, n$, $G_i(R_1;R_2) = G_i(R_1) \cup G_i(R_2)$;

(4) $\Phi(R^*) = (\Delta;\Phi(R))^*$,

$$G_i(R^*) = \begin{cases} G_i(R) \cup \{\Delta\} & \text{if } A_R \cap A_i \neq \emptyset \\ G_i(R) & \text{otherwise} \end{cases}$$

for $i = 1, 2, \ldots, n$ where Δ is any element of G such that $\Delta \notin A \cup G_i(R)$.

The synchronized p-concurrent regular expression Π(PCR) is defined as follows: Π(PCR): $[\Phi(R), \{B_1, B_2, \ldots, B_n\}]$ where $B_i = A_i \cup G_i(R)$, for $i = 1, 2, \ldots, n$.

From the above definition it immediately follows that Π(PCR) is a synchronized expression.

Corollary 3.8. *For every p-concurrent* ε*-free regular expression* PCR *we have* $\Pi(\mathrm{PCR}) \in \Omega$.

For any $V \subseteq \mathrm{vect}(A^*)$ and any $B \subseteq V$ let $V|B$ denote the concealment of actions from B in V. Corollary 3.9 follows immediately from Theorem 3.6 and Corollary 3.8.

Corollary 3.9. *For every p-concurrent* ε*-free regular expression* PCR: $[R, \{A_1, A_2, \ldots, A_n\}]$,

$$\mathrm{H}(\mathrm{PCR}) = \mathrm{H}(\Pi(\mathrm{PCR})) \mid G(R) \text{ and } \mathrm{RH}(\mathrm{PCR}) = \mathrm{RH}(\Pi(\mathrm{PCR})) \mid G(R).$$

Therefore, for a specification S of a sequential system in the form of a p-concurrent regular expression, we can first apply the transformation Π to obtain a proper specification $\Pi(\mathrm{S})$, and then the transformation $\mathbb{C}$ to get an equivalent concurrent specification $\mathrm{CS} \in \mathbb{C}(\Pi(\mathrm{S}))$.

Example 3.10. For the expression PCR_8: $[((a,b),c), \{A_1 = \{a\}, A_2 = \{b\}, A_3 = \{c\}\}]$ from Example 3.2 we would obtain $\Pi(\mathrm{PCR}_8)$: $[((a,(\Delta_1;b)),(\Delta_2;c)), \{A_1 = \{a,\Delta_1,\Delta_2\}, A_2 = \{b,\Delta_1,\Delta_2\}, A_3 = \{c,\Delta_2\}\}]$ and $(a,\Delta_1),\Delta_2 \,\|\, \varepsilon,(\Delta_1;b),\Delta_2 \,\|\, (\varepsilon,\varepsilon),(\Delta_2;c) \in \mathbb{C}(\Pi(\mathrm{PCR}_8))$.

Now we show the application of our technique to the well-known dining philosophers problem, first described by E.W. Dijkstra [10]. The major difficulty with any concurrent specification of this problem is to avoid deadlock. Here we present a sequential specification of the dining philosophers problem, which, using our technique, can be *automatically* transformed to the equivalent concurrent specification. Thus, the resulting concurrent specification is deadlock and starvation free. Our sequential specification of the dining philosophers problem comes from the very natural description of activities of each philosopher. Note that this example shows that depending on the initial form of a sequential specification, the resulting concurrent specification may provide a varying amount of concurrency.

Example 3.11. There are five philosophers who either eat spaghetti or think. When they become hungry, they sit at the circular table that has five chairs around it. There are five forks laid out on the table, thus there is precisely one fork between every adjacent two chairs. Since the spaghetti is tangled, a philosopher must have two forks to eat: one on the left and one on the right. If the philosopher cannot get two forks, he or she must wait until they are available. The forks are picked up one at a time and the left fork is picked up first. Using self-explanatory names to denote actions, the sequential specification of the ith philosopher can be written as: $\mathrm{pick\_left}_i;\mathrm{pick\_right}_i;\mathrm{eat}_i;\mathrm{put\_left}_i;\mathrm{put\_right}_i$. The sequential specification S_1 of the

problem has now the following form:

$((\text{pick\_left}_1;\text{pick\_right}_1;\text{eat}_1;\text{put\_left}_1;\text{put\_right}_1)^*,$

$(\text{pick\_left}_2;\text{pick\_right}_2;\text{eat}_2;\text{put\_left}_2;\text{put\_right}_2)^*,$

$(\text{pick\_left}_3;\text{pick\_right}_3;\text{eat}_3;\text{put\_left}_3;\text{put\_right}_3)^*,$

$(\text{pick\_left}_4;\text{pick\_right}_4;\text{eat}_4;\text{put\_left}_4;\text{put\_right}_4)^*,$

$(\text{pick\_left}_5;\text{pick\_right}_5;\text{eat}_5;\text{put\_left}_5;\text{put\_right}_5)^*)^*.$

Clearly, the only actions that are in mutual conflict are those that operate on the *same* fork. The sets of conflicting actions are listed below:

fork-1: $\{\text{pick\_left}_1,\text{eat}_1,\text{put\_left}_1,\text{pick\_right}_2,\text{eat}_2,\text{put\_right}_2\},$

fork-2: $\{\text{pick\_left}_2,\text{eat}_2,\text{put\_left}_2,\text{pick\_right}_3,\text{eat}_3,\text{put\_right}_3\},$

fork-3: $\{\text{pick\_left}_3,\text{eat}_3,\text{put\_left}_3,\text{pick\_right}_4,\text{eat}_4,\text{put\_right}_4\},$

fork-4: $\{\text{pick\_left}_4,\text{eat}_4,\text{put\_left}_4,\text{pick\_right}_5,\text{eat}_5,\text{put\_right}_5\},$

fork-5: $\{\text{pick\_left}_5,\text{eat}_5,\text{put\_left}_5,\text{pick\_right}_1,\text{eat}_1,\text{put\_right}_1\}.$

From Definition 2.15, we obtain the following concurrent specification $S_2 \in \mathbb{C}(S_1)$:

$((\text{pick\_left}_1;\text{eat}_1;\text{put\_left}_1)^*,(\text{pick\_right}_2;\text{eat}_2;\text{put\_right}_2)^*,\varepsilon,\varepsilon,\varepsilon)^* \|$

$(\varepsilon,(\text{pick\_left}_2;\text{eat}_2;\text{put\_left}_2)^*,(\text{pick\_right}_3;\text{eat}_3;\text{put\_right}_3)^*,\varepsilon,\varepsilon)^* \|$

$(\varepsilon,\varepsilon,(\text{pick\_left}_3;\text{eat}_3;\text{put\_left}_3)^*,(\text{pick\_right}_4;\text{eat}_4;\text{put\_right}_4)^*,\varepsilon)^* \|$

$(\varepsilon,\varepsilon,\varepsilon,(\text{pick\_left}_4;\text{eat}_4;\text{put\_left}_4)^*,(\text{pick\_right}_5;\text{eat}_5;\text{put\_right}_5)^*)^* \|$

$((\text{pick\_right}_1;\text{eat}_1;\text{put\_right}_1)^*,\varepsilon,\varepsilon,\varepsilon,(\text{pick\_left}_5;\text{eat}_5;\text{put\_left}_5)^*)^*$

Unfortunately, the above solution is *not* correct, because all philosophers can pick up the left fork simultaneously, after which they are *deadlocked* (see [19]). Now we consider the synchronized p-concurrent expression $S_{1'} = \Pi(S_1)$:

$(\Delta_0;((\Delta_{10};\text{pick\_left}_1;\text{pick\_right}_1;\text{eat}_1;\text{put\_left}_1;\text{put\_right}_1)^*),$

$(\Delta_1;(\Delta_{20};\text{pick\_left}_2;\text{pick\_right}_2;\text{eat}_2;\text{put\_left}_2;\text{put\_right}_2)^*),$

$(\Delta_2;(\Delta_{30};\text{pick\_left}_3;\text{pick\_right}_3;\text{eat}_3;\text{put\_left}_3;\text{put\_right}_3)^*),$

$(\Delta_3;(\Delta_{40};\text{pick\_left}_4;\text{pick\_right}_4;\text{eat}_4;\text{put\_left}_4;\text{put\_right}_4)^*),$

$(\Delta_4;(\Delta_{50};\text{pick\_left}_5;\text{pick\_right}_5;\text{eat}_5;\text{put\_left}_5;\text{put\_right}_5)^*)))^*.$

Above, Δ_0 guards the outer loop, for $i = 1, \ldots, 5$, Δ_{i0} guard inner loops, while for $i = 1, \ldots, 4$, Δ_i guard alternatives. The sets of depending actions are listed below.

fork-1: $\{\text{pick\_left}_1,\text{eat}_1,\text{put\_left}_1,\text{pick\_right}_2,\text{eat}_2,\text{put\_right}_2,\Delta_0,\Delta_1,\Delta_2,\Delta_3,\Delta_4,\Delta_{10},\Delta_{20}\},$

fork-2: $\{\text{pick\_left}_2,\text{eat}_2,\text{put\_left}_2,\text{pick\_right}_3,\text{eat}_3,\text{put\_right}_3,\Delta_0,\Delta_1,\Delta_2,\Delta_3,\Delta_4,\Delta_{20},\Delta_{30}\},$

fork-3: $\{\text{pick\_left}_3,\text{eat}_3,\text{put\_left}_3,\text{pick\_right}_4,\text{eat}_4,\text{put\_right}_4,\Delta_0,\Delta_2,\Delta_3,\Delta_4,\Delta_{30},\Delta_{40}\},$

fork-4: $\{\text{pick\_left}_4,\text{eat}_4,\text{put\_left}_4,\text{pick\_right}_5,\text{eat}_5,\text{put\_right}_5,\Delta_0,\Delta_3,\Delta_4,\Delta_{40},\Delta_{50}\},$

fork-5: $\{\text{pick\_left}_5,\text{eat}_5,\text{put\_left}_5,\text{pick\_right}_1,\text{eat}_1,\text{put\_right}_1,\Delta_0,\Delta_4,\Delta_{10},\Delta_{50}\}.$

Thus, the concurrent specification $S_3 \in \mathbb{C}(\Pi(S_1))$ has the following form:

$(\Delta_0;(\Delta_{10};\text{pick\_left}_1;\text{eat}_1;\text{put\_left}_1)^*,(\Delta_1;(\Delta_{20};\text{pick\_right}_2;\text{eat}_2;\text{put\_right}_2)^*),\Delta_2,\Delta_3,\Delta_4)^* \|$

$(\Delta_0;(\varepsilon,(\Delta_1;(\Delta_{20};\text{pick\_left}_2;\text{eat}_2;\text{put\_left}_2)^*),(\Delta_2;(\Delta_{30};\text{pick\_right}_3;\text{eat}_3;\text{put\_right}_3)^*),\Delta_3,\Delta_4)^* \|$

$(\Delta_0;(\varepsilon,\varepsilon,(\Delta_2;(\Delta_{30};\text{pick\_left}_3;\text{eat}_3;\text{put\_left}_3)^*),(\Delta_3;(\Delta_{40};\text{pick\_right}_4;\text{eat}_4;\text{put\_right}_4)^*),\Delta_4)^* \|$

$(\Delta_0;(\varepsilon,\varepsilon,\varepsilon,(\Delta_3;(\Delta_{40};\text{pick\_left}_4;\text{eat}_4;\text{put\_left}_4)^*),(\Delta_4;(\Delta_{50};\text{pick\_right}_5;\text{eat}_5;\text{put\_right}_5)^*))^* \|$

$(\Delta_0;(\varepsilon,\varepsilon,\varepsilon,(\Delta_4;(\Delta_{50};\text{pick\_left}_5;\text{eat}_5;\text{put\_left}_5)^*),(\Delta_{10};\text{pick\_right}_1;\text{eat}_1;\text{put\_right}_1)^*)^*.$

Now, not all the philosophers can pick up the left fork at the same time, because the synchronization guards are preventing them from doing so, hence, the deadlock has been avoided. Actually, from the above theory it follows that the resulting concurrent specification S_3 is always *adequate*, i.e. for every history H and every action x, this action can be executed in a certain continuation of history H. Clearly, adequacy implies that none of the philosophers may ever starve. But, although correct, our solution limits the amount of concurrency since at most one of the philosophers can be eating at any particular time. In [37], it was shown that a different sequential specification will result in a "more" concurrent solution: If we specify S_1 as: (phil-1;phil-2;phil-3;phil-4;phil-5)* or, in a fully expanded form:

$((\text{pick\_left}_1;\text{pick\_right}_1;\text{eat}_1;\text{put\_left}_1;\text{put\_right}_1)^*;$

$(\text{pick\_left}_2;\text{pick\_right}_2;\text{eat}_2;\text{put\_left}_2;\text{put\_right}_2)^*;$

$(\text{pick\_left}_3;\text{pick\_right}_3;\text{eat}_3;\text{put\_left}_3;\text{put\_right}_3)^*;$

$(\text{pick\_left}_4;\text{pick\_right}_4;\text{eat}_4;\text{put\_left}_4;\text{put\_right}_4)^*;$

$(\text{pick\_left}_5;\text{pick\_right}_5;\text{eat}_5;\text{put\_left}_5;\text{put\_right}_5)^*)^*,$

then we would arrive at

$(\Delta_0;(\Delta_{10};\text{pick\_left}_1;\text{eat}_1;\text{put\_left}_1)^*;(\Delta_{20};\text{pick\_right}_2;\text{eat}_2;\text{put\_right}_2)^*)^* \|$

$(\Delta_0;(\Delta_{20};\text{pick\_left}_2;\text{eat}_2;\text{put\_left}_2)^*;(\Delta_{30};\text{pick\_right}_3;\text{eat}_3;\text{put\_right}_3)^*)^* \|$

$(\Delta_0;(\Delta_{30};\text{pick\_left}_3;\text{eat}_3;\text{put\_left}_3)^*;(\Delta_{40};\text{pick\_right}_4;\text{eat}_4;\text{put\_right}_4)^*)^* \|$

$(\Delta_0;(\Delta_{40};\text{pick\_left}_4;\text{eat}_4;\text{put\_left}_4)^*;(\Delta_{50};\text{pick\_right}_5;\text{eat}_5;\text{put\_right}_5)^*)^* \|$

$(\Delta_0;(\Delta_{10};\text{pick\_right}_1;\text{eat}_1;\text{put\_right}_1)^*;(\Delta_{50};\text{pick\_left}_5;\text{eat}_5;\text{put\_left}_5)^*)^*,$

which is truly concurrent, as two philosophers (not arbitrary two, of course) are allowed to eat at the same time.

In the next section we show some practical applications of the theory developed above.

4. Banach programming language

To explain our ideas let us recall a well-known algorithm which performs the copying of records from the input queue to the output queue (see [5]).

Example 4.1. Consider the following three basic actions:

get	to get the next record from the input queue and store it in the input buffer;
copy	to copy the input buffer to the output buffer;
put	to put the output buffer to the output queue.

The sequential specification of a system that performs copying is

```
for i := 1 to length(input_queue) do begin
  get;
  copy;
  put;
end;
```

In order to explicitly program a concurrent execution of the above algorithm, independent actions must be consecutive so that the concurrent composition can be applied to them. Thus the above specification will have to be transformed to the equivalent sequential specification:

```
get;
for i := 1 to length(input_queue) do begin
  copy;
  put;
  get;
end;
copy;
put;
```

(here, for simplicity, we assume that the input queue is not empty), and finally, using the well-known **cobegin-coend** program construct (see [5]), the concurrent specification can be given:

```
get;
for i := 1 to length(input_queue) do begin
  copy;
  cobegin
    put;
    get;
  coend;
end;
copy;
put;
```

In this example, the transformation from the first sequential specification to the second specification is done in *preparation* for a concurrent specification; the operations get and put are disjoint, or *independent* and so they can be executed concurrently.

As Brinch Hansen points out in [5], it is fairly easy to make a mistake even in this simple case. For example, placing **cobegin** before the copy action would result in incorrect computations. The problem is that the user is forced to take care of low-level sychronization details. Clearly, for other programs such as sorting programs, matrix manipulations, etc., such synchronization details can be much more sophisticated than in the above example, but for such programs concurrency is used only to improve efficiency. In our approach the user is responsible for declaring which actions are independent, and so is relieved from having to consider synchronization details. Thus, the user provides what we call a *complete specification*: a pure sequential specification *and* a declaration of independency. (Using the formalism introduced in the previous section, a complete specification is a certain representation of a p-concurrent regular expression.) For the copying program, the complete specification consists of the original sequential specification, which in this case does not have to be modified, and a declaration of independent actions:

```
{complete sequential specification}
for i := 1 to length(input_queue) do begin
  get;
  copy;
  put;
end;
{declaration of independency}
independency
  ind(get, put)
end;
```

In our approach, to obtain a concurrent version of the *existing* software, program sections that are independent have to be specified. This may first require some modularization of the program, i.e. an encapsulation of such program sections in structured data types (see the following examples). Clearly, we need a programming language to write programs as above. We call this language Banach.[1] Banach is a Pascal-like language with the four additional constructs: a nondeterministic choice operator, a declaration of events, the REP statement, and a declaration of independency. Here, REP stands for a REPLICATOR, and is very much the same as the replicator in occam [15], Macro COSY [31], and the divide-in operator of [32, 33]. The basic unit of a computation is called an *event*, which is executed as atomic action. Events represent structured data types, much like packages in the Ada programming language [40]. Events can be activated in the same way as Pascal procedures are called, or Simula 67's class objects [3] are generated: the occurrence of the event name with the list of actual parameters, if any, in the instruction section of the program denotes an activation of the event and execution of its body. A Banach program consists of three sections; the declaration section, the instruction

[1] Stefan Banach (1892-1945), Polish mathematician who developed the theory of functional analysis.

section, and the independency section. The latter section starts with the keyword **independency** and declares which events are independent, using another keyword **ind**. This section may contain the declaration of local variables, followed by the clause, when BooleanExpression ind(event(...), event(...)). The BooleanExpression determines the domain of parameters for which the invocations of the events are to be independent. In Banach, there are two types of parameters of events: passed by value and passed by value-result (as in Ada). We require the following.

Requirement 4.2. (1) No actual parameters passed by value-result can be shared by two, or more independent events.

(2) Events cannot be textually nested.

(3) Independency can be declared for events only.

(4) In the main program, every primitive action, such as an assignment, or a Boolean expression, must be an event activation.

(5) No recursion of events is allowed (as in occam [15]).

These may not be the weakest requirements needed to preserve the equivalence of the sequential and Banach specification. (See [19] for the discussion of more general rules.) The reason for Requirement 4.2(1) is that for parameters passed by value-result on entry to the event body, the formal parameter gets the value of the actual parameter and then on exit, the actual parameter gets the value of the formal parameter. Thus, sharing the actual parameter may result in ill-defined concurrent access to the same variable. If concurrent execution is allowed, events are *atomic* actions, that is the execution of the event's body consists of a single action and cannot be divided into a number of concurrently executed subactions. For this reason we require 4.2(2) and 4.2(3). From the theory given in Section 3 of this paper, it follows that the Banach main program must give rise to the sequential specification in the form of the sequence of events. Thus we additionally require 4.2(4).

Now we give several examples of Banach programs.

Example 4.3. Consider the following implementation of the copying algorithm from Example 4.1. We assume that we have the operations available: input(f) to read in input data, length(Q) to compute the length of the queue, into(x, Q) to append x to the queue Q, and out(x, Q) to remove the first element of Q and return it in x.

```
Program Copying;
Var i:    integer;
  f, g:   QUEUE;
  x, y:   T;
```

```
event put (x: T; result f: QUEUE);
begin
  into (x, f); (* append x to f *)
end;
event get (result x: T; result f: QUEUE);
begin
  out (x, f); (* remove first *)
end;
event copy (x: T; result y: T);
begin
  y := x;
end;
begin
  input (f); (* read input data *)
  for i := 1 to length(f) do begin
    get (x, f);
    copy (x, y);
    put (y, g);
  end;
  independency
    ind(get, put)
  end;
end.
```

The above declaration of independent events does not provide parameters of the events get and put, which means that for *any* values of parameters these events are independent. If the **for** statement were implemented in terms of the **while** statement, for length(f)=3, the above code would be expanded to the following sequence of actions:

$$\text{input(f)};\ \text{get}_0(x, f);\ i := 1;\ \text{aux} := \text{length(f)};\ i <= \text{aux};\ \text{copy}_1(x, y);$$
$$\text{get}_1(x, f);\ \text{put}_1(y, g);\ i := i+1;\ i <= \text{aux};\ \text{copy}_2(x, y);\ \text{get}_2(x, f);$$
$$\text{put}_2(y, g);\ i := i+1;\ i <= \text{aux};\ \text{copy}_3(x, y);\ \text{get}_3(x, f);\ \text{put}_3(y, g);$$
$$i := i+1;\ i <= \text{aux};\ \text{copy}_4(x, y);\ \text{put}_4(y, g);$$

where aux is an auxiliary variable and indices in the calls to copy, get and put are used for readability only.

To satisfy Requirement 4.2(4) one would have to declare many events, and then activate them in the main program. To avoid having to go through this rather cumbersome procedure we introduce *anonymous events*. Since these events do not have names, they cannot be declared as independent; therefore anonymous events are always in conflict. Anonymous events can be explicitly specified by the programmer by enclosing statements in angle brackets ⟨...⟩, or they will be implicitly declared by the implementation. In the latter case, the implementation will group the longest sequences of statements which are *not* independent into single events.

(This is because events will have processors allocated, and so we want to avoid unnecessary events.) For the above example:

$\langle$input (f)$\rangle$; $get_0(x, f)$; $\langle i := \rangle$; $\langle aux := length(f); i <= aux\rangle$;
$copy_1(x, y)$; $get_1(x, f)$; $put_1(y, g)$; $\langle i := i+1; i <= aux\rangle$; $copy_2(x, y)$;
$get_2(x, f)$; $put_2(y, g)$; $\langle i := i+1; i <= aux\rangle$; $copy_3(x, y)$; $get_3(x, f)$;
$put_3(y, g)$; $\langle i := i+1; i <= aux\rangle$; $copy_4(x, y)$; $put_4(y, g)$;

From the above discussion it follows that for the copying program there is no need to declare the copy action as a named event (since it does not appear in the declaration section of independency), but in view of Requirement 4.2(3) we need to declare put and get as named events because we wish to specify them as independent.

Clearly, the above sequence of event occurrences can be interpreted as a *total* order. The independency relation relaxes some sequentialization and transforms this total order into the *partial* order shown in Fig. 9, which will allow us to execute this program concurrently. The copy action is now enclosed in angle brackets because

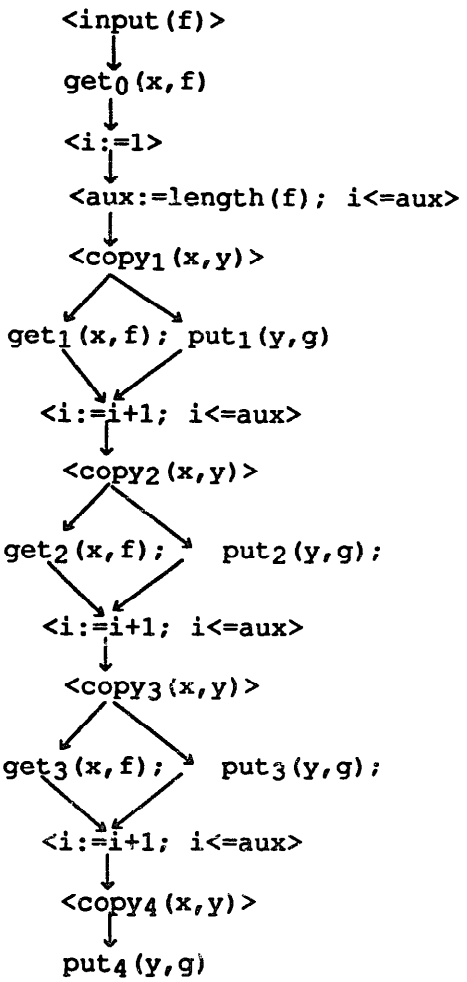

Fig. 9.

it is not a named event anymore. Events with no order among them can be executed concurrently. Strictly speaking, in the sequence:

$$get_1(x, f);\ put_1(y, g);\ \langle i := i+1;\ i <= aux\rangle;$$

all three actions are independent. They will *not* be subject to potential concurrent execution as the event $\langle i := i+1;\ i <= aux\rangle$; is an unnamed event and we need Requirement 4.2(3). If the programmer were willing to rewrite the for statement and declare an event that forms the above two substitutions, then all three actions could be executed concurrently.

Another solution to the above problem is to use *replicators* instead of the for statements. In general, the syntax of REP is **REP** i := exp1 **BY** exp2 **UNTIL** exp3 **DO** stat. **REP** differs from the **for** statement in that it avoids the implicit update of the control variable. For example: **REP** i := 1 **BY** i + 1 **UNTIL** i <= n stat(i) denotes

stat(1); stat(2); · · ·; stat(n).

Using the replicator, the main program in the above Copying program would be rewritten as follows:

```
REP i := 1 BY i + 1 UNTIL length(f) DO begin
  get(x, f);
  copy(x, y);
  put(y, g);
end;
```

(for another application of the REP construct to Shellsort program, see [21]).

Example 4.4. The program shown below computes the quotient and the remainder of two integers:

```
program division;
var a,b,q,r: integer;
event incr (result q: integer);
begin
  q := q + 1;
end;
event decr (value b: integer; result r: integer);
begin
  r := r - b;
end;
  begin
    input (a, b); t := a; q := 0;
    while r >= b do begin
      decr(b, r);
      incr(q);
    end;
```

```
    output(q, r);
  end;
  independent
      ind(incr, decr)
  end;
end.
```

It should be noted that if we switched the two statements that form the body of the above loop we would obtain:

```
while r>=b do begin
  incr(q);
  decr(b, r);
end;
```

Now, if we declare the event test (r, b) to test the stop condition, we could declare two independent events, test and incr. The p-concurrent regular expression corresponding to this specification is not proper, but applying the transformation Π we obtain a proper expression.

Example 4.5. Our last example shows Banach's version of matrix multiplication algorithm. The scalar product of the ith row of the matrix a and the jth column of the array b is declared as an event.

```
program matrix;
const n=100;
type T=array[1..n] of array[1..n] of real;
Var a,b,c:T; i,j,k: integer;
event scalar (value i,j,k: integer);
begin
  c[i, j] := c[i, j] + a[i, k] * b[k, j];
end;
begin
  input (a, b);
  REP i:= 1 BY i+1 UNTIL i<=n DO
  REP j:= 1 BY j+1 UNTIL j<=n DO
  REP k:= 1 BY k+1 UNTIL k<=n DO
  scalar(i,j,k);
  output (c);
  independent
    x, x1, y, y1, z, z1: integer;
    when (x<>x1 or y<>y1)
    ind(scalar(x,y,z), scalar(x1, y1, z1))
  end;
end.
```

The above specification gives rise to a maximally concurrent specification, and a speed-up is significant: from n^3 *to* n. For example, assume that $n = 2$, and consider the expansion of the above REP statements:

scalar(1,1,1); scalar(1,1,2); scalar(1,2,1); scalar(1,2,2); scalar(2,1,1);

scalar(2,1,2); scalar(2,2,1); scalar(2,2,2).

The declaration of independent events can be translated to the following specification in terms of p-concurrent regular expressions (for details, see Section 5): A_1 = {scalar(1,1,1), scalar(1,1,2)}, A_2 = {scalar(1,2,1), scalar(1,2,2)}, A_3 = {scalar(2,1,1), scalar(2,1,2)}, A_4 = {scalar(2,2,1), scalar(2,2,2)}. Thus, after we apply the transformation $\mathbb{C}$ we obtain

(scalar(1,1,1);scalar(1,1,2)) || (scalar(1,2,1);scalar(1,2,2) ||

(scalar(2,1,1);scalar(2,1,2)) || (scalar(2,2,1);scalar(2,2,2)).

5. Modelling Banach programs by p-concurrent regular expressions

In this section we show how p-concurrent synchronized regular expressions can be used to model the control structure and synchronization properties of complete sequential specifications, similar to those written in Banach programming language. The idea is not new, it goes back to the early 1970s when it was shown that regular expressions (in general, formal grammars) can be used to model the control structure of various kinds of programs, see e.g. [11, 34] for modelling Algol-like programs. Banach programming language is specially designed to use the above approach, but our approach can be applied to most programming languages.

The basic element of a Banach program is an atomic *event*, and due to existence of unnamed events, the first part of every Banach program, without a declaration of independent event, can be viewed as a certain generator of sequences of events. Since events are atomic and no recursion is allowed, these sequences of events are entirely generated by the following operators:

for, REP, while-do, if-then-else, ";", "∨"

(the latter operator denotes nondeterministic choice). The meaning of for is defined in terms of while-do, so in fact we have to consider only four operators in detail: while-do, if-then-else, ";", "∨". Let us now introduce names of main syntactic units in Banach-like languages.

Definition 5.1. Let us denote by: *E*, event; *NE*, named event; *UNE*, unnamed event; *C*, command; *I*, definition of independency relation; *D*, declaration; *SQS*, sequential specification; *BLP*, Banach-like program. The essential part of an abstract syntax

for Banach like languages can be defined as follows (we omit I/O commands):

BLP ::= **program** *name*; SQS;I **end**

SQS ::= D **begin** C **end**

C ::= $E \mid C;C \mid C \vee C \mid$ **while** E **do** $C \mid$ **if** E **then** C **else** C
$\mid$ **for**... $\mid$ REP...

E ::= $NE \mid UNE$.

Now, we define the mapping from Banach programs to p-concurrent regular expressions.

Definition 5.2. Let ψ be a mapping from the sequential specification part of Banach-like programs into the set of regular expressions over an alphabet A, defined as follows:

$\psi(SQS) = \psi(C)$.

$\psi(E) = e$, here, $e \in A$ is uniquely associated with the event E.

$\psi(C_1;C_2) = \psi(C_1);\psi(C_2)$.

$\psi(C_1 \vee C_2) = \psi(C_1),(\Delta;\psi(C_2))$.

$\psi($**while** E **do** $C) = (\Delta;e;\psi(C))^*;e'$.

$\psi($**if** E **then** C_1 **else** $C_2) = (e;\psi(C_1)),(\Delta;e';\psi(C_2))$.

In the two above expressions, $\psi(E) = e$, $e' \in A$ is associated with the event "not E", and Δ is a synchronization guard, as defined in Section 4.

It is easy to see that every expression of the form $\psi(SQS)$ is a well-defined, synchronized, p-concurrent regular expression. Now, we consider an example of the appliction of the mapping ψ.

Example 5.3. The following Banach program computes the sum of two integers using the successor and predecessor operations:

```
program addition;
(* sequential specification SQS *)
(* declaration section D *)
var x, y:integer;
event xgt0:Boolean;
begin
  xgt0 := x > 0;
```

```
end;
event xminus1;
begin
  x:=x-1;
end;
event yplus1;
begin
  y:=y+1;
end; (* of D *)
(* commands C *)
begin
  input(x, y);
  while xgt0 do begin
    xminus1;
    yplus1;
  end;
  output (y);
end; (* of C *)
(* independency declaration I *)
independency
  ind (xminis1, yplus1);
end; (* of I *)
(* end of SQS *)
end.
```

For the above program, ignoring I/O, we have $\psi(SQS) = (\Delta; xgt0; xminus1; yplus1)^*; Not xgt0$, where $Not xgt0$ is a name of a Banach even equivalent to "**not** $xgt0$".

The second part of Banach-like specification, i.e. the declaration of independency relation I, can be easily modelled by a symmetric and irreflexive relation $ind \subseteq A \times A$. Now we show that the distribution $\{A_1, A_2, \ldots, A_n\}$ used in the definition of a p-concurrent regular expression can be derived from the pair (A, ind), where ind is a symmetric and irreflexive relation over A. If $dep = A \times A - ind$, then dep is a symmetric and reflexive relation which can be interpreted as a dependency relation. Let us define the set $COV_{ind} \subseteq 2^{2^A}$ of all families of coverings $\{A_1, A_2, \ldots, A_n\}$ of A as follows: $\{A_1, A_2, \ldots, A_n\} \in COV_{ind}$ iff $A = A_1 \cup A_2 \cup \cdots \cup A_n \;\&\; (\forall a, b \in A)(a, b) \in ind \Leftrightarrow ((\forall_i) a \notin A_i \vee b \notin A_i)$. Any element of COV_{ind} can be used as the distribution $\{A_1, A_2, \ldots, A_n\}$, in particular, one can use the family DEP of all dependent sets defined by: $A \in DEP \Leftrightarrow (\forall a, b \in A)(a, b) \in dep$, or a family MDEP of all maximal dependent sets, defined by $A \in MDEP \Leftrightarrow A \in DEP \;\&\; ((\forall B \in DEP) A \subseteq B$ implies $A = B)$. In terms of graph theory, DEP is the set of all cliques, i.e. complete subgraphs of the graph of the relation dep, and MDEP is the set of all maximal cliques of that graph.

Appendix

Proof of Theorem 3.6. First, we introduce some auxiliary notations (borrowed from [19]). Let $A_1, A_2, \ldots, A_n$ be alphabets, $A = A_1 \cup A_2 \cup \cdots \cup A_n$, ind $\subset A \times A$ be a relation from Definition 2.7, vect: $2^{A^*} \to 2^{A_1^* \times A_2^* \times \cdots \times A_n^*}$ be the mapping from Definition 2.5, and finally let $\underline{\text{vect}}: 2^{A^*} \to 2^{A_1^* \times A_2^* \times \cdots \times A_n^*}$ be the following mapping:

$$\underline{\text{vect}}(L) = h_1(L) \times h_2(L) \times \cdots \times h_n(L) \cap \text{vect}(A^*).$$

From the above definitions we have the following (compare [19]).

Corollary A.1

(1) $\underline{\text{vect}}(L) = \{\underline{x}: (\exists x \in A^*)\ (\forall i = 1, 2, \ldots, n)\ h_i(x) \in h_i(L)\}$,

(2) $\text{vect}(L) \subseteq \underline{\text{vect}}(L)$,

(3) $\text{vect}(\text{Pref}(L)) \subset \text{Pref}(\text{vect}(L))$,

(4) $\text{Pref}(\underline{\text{vect}}(L)) \subset \underline{\text{vect}}(\text{Pref}(L))$,

(5) $\underline{\text{vect}}(L_1)\underline{\text{vect}}(L_2) \subset \underline{\text{vect}}(L_1 L_2)$,

(6) $\underline{\text{vect}}(L_1) \cup \underline{\text{vect}}(L_2) \subset \underline{\text{vect}}(L_1 \cup L_2)$.

Now, we rephrase Theorem 3.6 in an equivalent form which is easier to prove.

Theorem A.2. *Let* PCR $= [R, \{A_1, A_2, \ldots, A_n\}] \in \Omega$, *that is* $R \in \Sigma$, *and let* $L = \mathcal{L}(R)$. *Then*

(1) $\text{Pref}(\text{vect}(L)) = \underline{\text{vect}}(\text{Pref}(L))$,

(2) $\text{vect}(L) = \underline{\text{vect}}(L)$.

Proof. The proof is by induction on the structure of expressions from the class Σ. Clearly, for expr $= a$, where a is an atomic event, both (1) and (2) are true. In the proof of the inductive step, we often identify regular expressions with languages generated by them. First, we prove the following lemma.

Lemma A.3

(1) $\underline{\text{vect}}((\Delta;\text{expr})^*) = (\{\underline{\Delta}\}\underline{\text{vect}}(\text{expr}))^*$,

(2) $\underline{\text{vect}}(\text{expr}_1, (\Delta;\text{expr}_2)) = \underline{\text{vect}}(\text{expr}_1) \cup \{\underline{\Delta}\}\underline{\text{vect}}(\text{expr}_2)$.

Proof. (1) From Corollary A.1(1) we have $\underline{vect}((\Delta;expr)^*) = \{\underline{x}: x \in A^*, h_i(x) \in h_i((\Delta;expr)^*)\}$. For $i \notin i(\Delta)$, $h_i(x) = \varepsilon$ and $h_i((\Delta;expr)^*) = \{\varepsilon\}$, thus $\{\underline{x}: x \in A^*, h_i(x) \in h_i((\Delta;expr)^*)\} = \{\underline{x}: x \in A^*, h_i(x) \in h_i((\Delta;expr)^*), i \in i(\Delta)\} = \{\underline{x}: x \in A^*, h_i(x) \in (\{\Delta\}h_i(expr)^*), i \in i(\Delta)\} = \{\underline{x}: x \in A^*, h_i(x) \in (\{\Delta\}h_i(expr)^{k_i}), i \in i(\Delta), k_i \geq 0\}$. Since $(\forall a \in A_{expr})\ i(a) \subseteq i(\Delta)$ we have $\forall i,j \in i(\Delta)$, $k_i = k_j$, so $\{\underline{x}: x \in A^*, h_i(x) \in (\{\Delta\}h_i(expr)^{k_i}), i \in i(\Delta), k_i \geq 0\} = \{\underline{x}: x \in A^*, h_i(x) \in (\{\Delta\}h_i(expr))^k, i \in i(\Delta), k \geq 0\} = \{\underline{\Delta y_1 \Delta y_2 \cdots \Delta y_k}: h_i(y_j) \in h_i(expr), i \in i(\Delta), j = 1, 2, \ldots, k, k \geq 0\} = (\{\underline{\Delta}\}\underline{vect}(expr))^*$.

(2) From Corollary A.1(1) we have $\underline{vect}(expr_1,(\Delta;expr_2)) = \{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1,(\Delta;expr_2))\}$. For $i \notin i(\Delta)$ we have $h_i(x) = \varepsilon$ and $h_i(expr_1,(\Delta;expr_2)) = \{\varepsilon\}$, thus $\{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1,(\Delta;expr_2))\} = \{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1,(\Delta;expr_2)), i \in i(\Delta)\} = \{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1) \cup \{\Delta\}h_i(expr_2), i \in i(\Delta)\}$. Since $(\forall a \in A_{expr_1} \cup A_{expr_2})\ i(a) \subseteq i(\Delta)$ so it is *not* possible that there is $x \in A^*$ such that $h_{i_1}(x) \in h_i(expr_1)$ and $h_{i_2}(x) \in h_i(expr_2)$ for some $i_1, i_2 \in i(\Delta)$. Therefore $\{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1) \cup \{\Delta\}h_i(expr_2), i \in i(\Delta)\} = \{\underline{x}: x \in A^*, h_i(x) \in h_i(expr_1), i \in i(\Delta)\} \cup \{\underline{\Delta y}: h_i(y) \in h_i(expr_2), i \in i(\Delta)\} = \underline{vect}(expr_1) \cup \{\underline{\Delta}\}\underline{vect}(expr_2)$. □

From Lemma A.3, it immediately follows that if $vect(expr) = \underline{vect}(expr)$ and for $i = 1, 2$ $vect(expr_i) = \underline{vect}(expr_i)$ then $vect((\Delta;expr)^*) = \underline{vect}((\Delta;expr)^*)$ and $vect(expr_1,(\Delta;expr_2)) = \underline{vect}(expr_1,(\Delta;expr_2))$. Now, we prove another lemma.

Lemma A.4

(1) $\underline{vect}(Pref((\Delta;expr)^*)) = \{\underline{\varepsilon}\} \cup \underline{vect}((\Delta;expr)^*)\{\underline{\Delta}\}\underline{vect}(Pref(expr))$,

(2) $\underline{vect}(Pref(expr_1,(\Delta;expr_2))) = \underline{vect}(Pref(expr_1)) \cup \{\underline{\Delta}\}\underline{vect}(Pref(expr_2))$.

Proof. The proof is similar to that of Lemma A.3. Since $i(a) \subseteq i(\Delta)$ we conclude that $h_i(x)$ belongs to the image under h_i of expressions that contain Δ's, which implies that $\underline{x}$ belongs to the image under $\underline{vect}$ of such expressions The details are left to the reader. □

From the definition of the mapping vect it follows that $Pref(vect((\Delta;expr)^*)) = \{\underline{\varepsilon}\} \cup vect((\Delta;expr)^*)\{\underline{\Delta}\}Pref(vect(expr))$, and $Pref(vect(expr_1,(\Delta;expr_2))) = Pref(vect(expr_1)) \cup \{\underline{\Delta}\}Pref(vect(expr_2))$. From Lemmas A.3 and A.4 it follows that if for $i = 1,2$, $vect(expr_i) = \underline{vect}(expr_i)$, and $Pref(vect(expr_i)) = \underline{vect}(Pref(expr_i))$ then $Pref(vect((\Delta;expr_1)^*)) = \underline{vect}(Pref((\Delta;expr_1)^*))$ and we have $Pref(vect(expr_1,(\Delta;expr_2))) = \underline{vect}(Pref(expr_1,(\Delta;expr_2)))$. Thus, Theorem A.2 is true for $R = a$, $R = (\Delta;expr)^*$, $R = (expr_1,(\Delta;expr_2))$. Now, we need two more lemmas.

Lemma A.5. *For every regular expression* expr (*not necessarily from the class* Σ) $\underline{\text{vect}}(\text{expr};a) = \underline{\text{vect}}(\text{expr})\{\underline{a}\}$.

Proof. From Corollary A.1(5) $\underline{\text{vect}}(\text{expr})\{a\} \subseteq \underline{\text{vect}}(\text{expr};a)$. Let $\underline{x} \in \underline{\text{vect}}(\text{expr};a) - \underline{\text{vect}}(\text{expr})\{\underline{a}\}$. Hence, for some $\underline{y} \notin \underline{\text{vect}}(\text{expr})$ we have $\underline{x} = \underline{ya}$. Since $\underline{y} \notin \underline{\text{vect}}(\text{expr})$ we have $(\exists i_0) h_{i_0}(y) \notin h_{i_0}(\text{expr})$, which implies that $h_{i_0}(ya) \notin h_{i_0}(\text{expr};a)$ and so $\underline{x} = \underline{ya} \notin \underline{\text{vect}}(\text{expr};a)$, which gives a required contradiction. □

Lemma A.6. *If, for* $\text{expr} \in \Sigma$, $\underline{\text{vect}}(\text{Pref}(\text{expr})) = \text{Pref}(\text{vect}(\text{expr}))$ *then for* $a \in A$, $\underline{\text{vect}}(\text{Pref}(\text{expr};a)) = \text{Pref}(\text{vect}(\text{expr};a))$.

Proof. Suppose that $\underline{\text{vect}}(\text{Pref}(\text{expr})) = \text{Pref}(\text{vect}(\text{expr}))$. From Corollary A.1 we have $\text{Pref}(\text{vect}(\text{expr},a)) \subset \underline{\text{vect}}(\text{Pref}(\text{expr};a))$. Let $\underline{x}$ be a shortest element (it may not be unique) of $\underline{\text{vect}}(\text{Pref}(\text{expr};a)) - \text{Pref}(\text{vect}(\text{expr};a))$. From the assumption of this lemma it follows that $\underline{x} = \underline{ya}$, where $\underline{x} \notin \text{Pref}(\text{vect}(\text{expr}))$, $\underline{y} \in \text{Pref}(\text{vect}(\text{expr}))$, and $(\forall v \in A^+) \underline{yav} \notin \text{Pref}(\text{vect}(\text{expr};a))$. This would mean that $\underline{\text{vect}}(\text{Pref}(\text{expr};a)) - \text{Pref}(\text{vect}(\text{expr};a)) = \underline{\text{vect}}(\text{Pref}(\text{expr};a)) - \text{vect}(\text{expr};a)$ and so the following conditions hold:

(i) $(\forall i) h_i(ya) \in h_i(\text{Pref}(\text{expr};a))$,

(ii) $(\forall w \in (\text{expr};a))\ (\exists i) h_i(w) \neq h_i(ya)$.

For every language $L \subseteq A^*$, $h_i(L) = \{h_i(z): z \in L\}$, hence $(\forall i)\ (\exists w_i \in \text{Pref}(\text{expr};a))$ such that $h_i(w_i) = h_i(ya)$. Since $\underline{ya} \notin \text{Pref}(\text{vect}(\text{expr}))$ we can assume that $w_i \in (\text{expr};a)$, for $i = 1, 2, \ldots, n$. The condition (ii) implies that $(\exists i_1, i_2)\ w_{i_1} \neq w_{i_2}$. From $(\text{expr};a) \in \Sigma$, it follows that $w_{i_1} = z\Delta v_1$, $w_{i_2} = zv_2$, $v_1, v_2 \in A^*$, $\#_\Delta(z) = 0$ (note that for the expressions in the class Σ, conflicts may occur only due to syntactic structures of the form $\text{expr}_1,(\Delta;\text{expr}_2)$ or $(\Delta;\text{expr}_1)^*;\text{expr}_2$.) Suppose that the conflict occurs due to the syntactic structure $(\Delta;\text{expr}_1)^*;\text{expr}_2$. Thus, $(\exists k \geq 1)\ (\forall i \in i(\Delta)\ h_i(ya) = z_i \Delta t_1^i \Delta t_2^i \cdots \Delta t_k^i r_i z_i')$, where $\#_\Delta(z_i) = 0$, $\Delta t_j^i \in h_i(\Delta;\text{expr}_1)$, for $i = 1, 2, \ldots, k-1$, $\Delta t_k^i \in h_i(\text{Pref}(\Delta;\text{expr}_1))$, $r_i \in h_i(\text{Pref}(\text{expr}_2))$, $\#_\Delta(h_i(ya)) = k$. Let $\alpha = \bigcup_{i \in i(\Delta)} \bigcup_{j=1}^{k} A_j^i$, where A_j^i stands for the alphabet of t_j^i. Assume that $\alpha = \{a_1, a_2, \ldots, a_r\}$ and let $s_j = \#_{a_j}(\Delta t_1^i \Delta t_2^i \cdots \Delta t_k^i)$, where i is any element from the sets $i(a_j)$, for $j = 1, 2, \ldots, r$ (note that s_j are well-defined.) Clearly, $\underline{ya} = \underline{x} = \underline{x_1 \Delta x_2}$, where $\#_\Delta(x_1) = 0$. Let x' be a string obtained from x by erasing first s_j occurrences of a_j from x_2, for $j = 1, 2, \ldots, r$, and k occurrences of Δ from Δx_2. Of course, $\underline{x}' = \underline{y'a}$, and $\underline{x}' \in \underline{\text{vect}}(\text{Pref}(\text{expr};a)) - \text{Pref}(\text{vect}(\text{expr};a))$, which is a contradiction because $\underline{x}'$ is shorter than $\underline{x}$ (as $k \geq 1$) and $\underline{x}$ has been chosen to be the shortest element. Suppose now that the conflict is due to the structure $\text{expr}_1,(\Delta;\text{expr}_2)$. We consider two cases: (a) $v_2 = \varepsilon$, and (b) $v_2 \neq \varepsilon$. Recall that $w_{i_1} = z\Delta v_1$ and $w_{i_2} = zv_2$.

(a) In this case $\varepsilon \in \text{expr}_1$, for example $\text{expr}_1 = (\Delta_1;\text{expr}_{11})^*;\cdots;(\Delta_k;\text{expr}_{1l})^*$. Using the same arguments as in the above case (i.e. for $(\Delta;\text{expr}_1)^*;\text{expr}_2$) we can erase from $\underline{ya} = \underline{x}$ both Δ and all symbol occurrences from $(\Delta;\text{expr}_2)$ in $\text{expr}_1,(\Delta;\text{expr}_2)$. The resulting $\underline{x}' = \underline{y'a}$ is shorter than $\underline{x}$ and belongs to the set $\underline{\text{vect}}(\text{Pref}(\text{expr};a)) - \text{Pref}(\text{vect}(\text{expr};a))$, which gives the required contradiction.

(b) Here, $w_{i_2} = zcv_{2'}$, where $c \in A$, $cv_{2'} \in \text{Pref}(\text{expr}_1)$, and $\Delta \notin i(c)$, which in contradiction with the definition of the class Σ. □

Note that in Lemma A.6. the assumption that $\text{expr} \in \Sigma$ is essential. Consider for example expr: (a,b), $A_1 = \{a,b\}$, $A_2 = \{a,c\}$, $A_3 = \{b,c\}$. Here, $\text{expr} \notin \Sigma$ and $\underline{\text{vect}}(\text{Pref}(a,b)) = \text{Pref}(\text{vect}(a,b)) = \{\underline{\varepsilon}, \underline{a}, \underline{b}\}$, but $\underline{\text{vect}}(\text{Pref}(a,b;c)) = \{\underline{\varepsilon}, \underline{a}, \underline{b}, \underline{c}, \underline{ac}, \underline{bc}\} \neq \{\underline{\varepsilon}, \underline{a}, \underline{b}, \underline{ac}, \underline{bc}\} = \text{Pref}(\text{vect}(a,b;c))$. From Lemmas A.5 and A.6 it follows that for $\text{expr} \in \Sigma$, if $\text{Pref}(\text{vect}(\text{expr})) = \underline{\text{vect}}(\text{Pref}(\text{expr}))$ and $\text{vect}(\text{expr}) = \underline{\text{vect}}(\text{expr})$ then $\text{Pref}(\text{vect}(\text{expr};a)) = \underline{\text{vect}}(\text{Pref}(\text{expr};a))$ and $\text{vect}(\text{expr};a) = \underline{\text{vect}}(\text{expr};a)$ so Theorem A.2 is true for R: expr;a. To complete the proof of the theorem we need two more lemmas.

Lemma A.7. *Let* $\text{expr};(\Delta;\text{expr}_1)^*$ *and* $\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)$ *belong to the class* Σ. *Then*

(1) $\underline{\text{vect}}(\text{expr};(\Delta;\text{expr}_1)^*) = \underline{\text{vect}}(\text{expr})\underline{\text{vect}}((\Delta;\text{expr}_1)^*)$,

(2) $\underline{\text{vect}}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)) = \underline{\text{vect}}(\text{expr})\underline{\text{vect}}(\text{expr}_1,(\Delta;\text{expr}_2))$.

Proof. (1) From Corollary A.1(5) we known that $\underline{\text{vect}}(\text{expr})\underline{\text{vect}}((\Delta;\text{expr}_1)^*) \subseteq \underline{\text{vect}}(\text{expr};(\Delta;\text{expr}_1)^*)$. Let $\underline{x} \in \underline{\text{vect}}(\text{expr};(\Delta;\text{expr}_1)^*)$, so

$$(\forall i) h_i(x) \in h_i(\text{expr}) h_i((\Delta;\text{expr}_1)^*).$$

For every $B \subseteq A$, let h^B, h_B: $A^* \to A^*$ be the following erasing homomorphisms:

$$(\forall a \in A) h_B(a) = \begin{cases} \varepsilon & \text{if } a \notin B, \\ a & \text{otherwise;} \end{cases} \qquad h^B(a) = \begin{cases} a & \text{if } a \notin B, \\ \varepsilon & \text{otherwise.} \end{cases}$$

If $\#_\Delta(x) = 0$ then $\underline{x} \in \underline{\text{vect}}(\text{expr}) \subseteq \underline{\text{vect}}(\text{expr})\underline{\text{vect}}((\Delta;\text{expr}_1)^*)$, so suppose that $\#_\Delta(x) \neq 0$. Clearly $x = x'\Delta x''$, $\#_\Delta(x') = 0$ and $\underline{x}' \in \text{Pref}(\underline{\text{vect}}(\text{expr}))$. Let $E = \{\Delta\} \cup \{a\colon\ i(a) \subseteq i(\Delta)\}$ and let $y = x'h^E(\Delta x'')h_E(\Delta x'')$. Note that $\underline{x} = \underline{y}$ and $\underline{y} \in \underline{\text{vect}}(\text{expr})\underline{\text{vect}}((\Delta;\text{expr}_1)^*)$.

(2) Again, from Corollary A.1(5) we have $\underline{\text{vect}}(\text{expr})\underline{\text{vect}}(\text{expr}_1,(\Delta;\text{expr}_2)^*) \subseteq \underline{\text{vect}}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_1))$. Let $\underline{x} \in \underline{\text{vect}}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_1))$. We have to consider two cases: (i) $\#_\Delta(x) = 1$ and (ii) $\#_\Delta(x) = 0$.

(i) $x = x'\Delta x''$, and $\underline{x}' \in \text{Pref}(\underline{\text{vect}}(\text{expr}))$. Let $E = \{\Delta\} \cup \{a\colon\ i(a) \subseteq i(\Delta)\}$ and let $y = x'h^E(\Delta x'')h_E(\Delta x'')$. Clearly, $\underline{x} = \underline{y}$ and $\underline{y} \in \underline{\text{vect}}(\text{expr})\underline{\text{vect}}(\Delta;\text{expr}_2)$.

(ii) Here, $(\forall i) h_i(x) \in h_i(\text{expr}) h_i(\text{expr}_1)$.

For every i, let $h_i(x) = y_i z_i$, where $z_i \in h_i(\text{expr}_1)$ and z_i is the longest such suffix of $h_i(x)$. Let us define $\alpha = \bigcup_{i \in i(\Delta)} A_{z_i}$ and assume that $\alpha = \{a_1, a_2, \ldots, a_r\}$. For every $j = 1, 2, \ldots, r$ let $k_j = \mu_{i \in i(\Delta)} \#_{a_j}(z_i)$ where μ stands for the minimum. Now, let x' be the string obtained from x by erasing all last k_j occurrences of a_j, for all $a_j \in \alpha$,

and let x'' be the string obtained from x by erasing all symbol occurrences except the last k_j occurrences of a_j, for all $a_j \in \alpha$. One can easily verify that both x' and x'' are well-defined and $\underline{x} = \underline{x'x''}$, while $\underline{x'} \in \underline{\text{vect}}(\text{expr})$ and $\underline{x''} \in \underline{\text{vect}}(\text{expr}_1)$ which ends the proof of the lemma. □

Lemma A.8. *Let* $\text{expr},\text{expr}_1,\text{expr}_2 \in \Sigma$, *and* $\underline{\text{vect}}(\text{Pref}(\text{expr})) = \text{Pref}(\text{vect}(\text{expr}))$. *Then*

(1) $\underline{\text{vect}}(\text{Pref}(\text{expr};(\Delta;\text{expr}_1)^*)) = \text{Pref}(\text{vect}(\text{expr};(\Delta;\text{expr}_1)^*))$,

(2) $\underline{\text{vect}}(\text{Pref}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2))) = \text{Pref}(\text{vect}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)))$.

Proof. (1) From Corollary A.1 it follows that $\text{Pref}(\text{vect}(\text{expr};(\Delta;\text{expr}_1)^*)) \subseteq \underline{\text{vect}}(\text{Pref}(\text{expr};(\Delta\cdot\text{expr}_1)^*))$. Let $\underline{x}$ be a shortest (it may not be unique) element of the set $\underline{\text{vect}}(\text{Pref}(\text{expr};(\Delta;\text{expr}_1)^*)) - \text{Pref}(\text{vect}(\text{expr};(\Delta;\text{expr}_1)^*))$. Since $\underline{\text{vect}}(\text{Pref}(\text{expr})) = \text{Pref}(\text{vect}(\text{expr}))$, we have $\underline{x} = \underline{y\Delta}$, $\underline{x} \notin \text{Pref}(\text{vect}(\text{expr}))$, $\underline{y} \in \text{Pref}(\text{vect}(\text{expr}))$, thus the following conditions hold:

(i) $(\forall i) h_i(y\Delta) \in h_i(\text{Pref}(\text{expr};\Delta)) \subseteq h_i(\text{Pref}(\text{expr};(\Delta;\text{expr}_1)^*)$,

(ii) $(\forall w \in (\text{expr};\Delta) \subseteq (\text{expr};(\Delta;\text{expr})^*))\ (\exists i) h_i(w) \neq h_i(y\Delta)$.

The remaining part of the proof of this case is almost identical to the proof of Lemma A.6 and is left to the reader.

(2) From Corollary A.1 it follows that $\text{Pref}(\text{vect}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2))) \subseteq \underline{\text{vect}}(\text{Pref}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_1)^*))$. Let $\underline{x}$ be a shortest (it may not be unique) element of the set $\underline{\text{vect}}(\text{Pref}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2))) - \text{Pref}(\text{vect}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)))$. Since $\underline{\text{vect}}(\text{Pref}(\text{expr})) = \text{Pref}(\text{vect}(\text{expr}))$, either $\underline{x} = \underline{y\Delta}$, $\underline{x} \notin \text{Pref}(\text{vect}(\text{expr}))$, and $\underline{y} \in \text{Pref}(\text{vect}(\text{expr}))$, or $\underline{x} = \underline{ya}$, for some $a \in A_{\text{expr}_1}$. But if $\underline{x} = \underline{ya}$ then one can easily prove that $\underline{x'} = \underline{y\Delta}$ also belongs to the set $\underline{\text{vect}}(\text{Pref}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2))) - \text{Pref}(\text{vect}(\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)))$, the lengths of x and x' are the same, and $\underline{x'} \notin \text{Pref}(\text{vect}(\text{expr}))$. The remaining part of the proof is almost the same as the proof of Lemma A.6 and is left to the reader. □

From Lemmas A.7 and A.8 it follows that Theorem A.2 is true for R: $\text{expr};(\Delta;\text{expr}_1)^*$ and R:$\text{expr};\text{expr}_1,(\Delta;\text{expr}_2)$, which completes the proof of the theorem. □

Acknowledgment

The work of the first author was partially supported by the NSERC grant OGP0036539 and the work of the second author was partially supported by the NSERC General Grant, Acadia University, 1989. The second author worked on this paper in the Department of Computer Science, the University of Western Ontario, London, Ontario, while on his sabbatical from Acadia University. The

authors would like to thank the anonymous referees and Teodor Rus for careful reading and helpful comments.

References

[1] R.J.R. Back, *A Method for Refining Atomicity in Parallel Algorithms*, Lecture Notes in Computer Science **366** (Springer, Berlin, 1989) 199-216.
[2] E. Best, *COSY: Its Relation to Nets and CSP*, Lecture Notes in Computer Science **255** (Springer, Berlin, 1986) 416-440.
[3] G. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nyggard, *Simula Begin* (Chartwell-Batt Ltd, 1979).
[4] P. Bose, Heuristic rule-based transformations for enhanced vectorization, in: *Proc. 17th Internat. Conf. on Parallel Processing*, Vol II (Pen. State Press, 1988) 63-66.
[5] P. Brinch Hansen, *Operating System Principles* (Prentice Hall, Englewood Cliffs, NJ, 1973).
[6] P. Brinch Hansen, Joyce—a programming language for distributed systems, *Software Practice and Experience* **17**(1) (1987) 29-50.
[7] P. Cartier and D. Foata, *Problemes Combinatoires de Commutation et Rearrangements*, Lecture Notes in Mathematics **85** (Springer, Berlin, 1969).
[8] D. Crookes and J.W.G. Elder, An experiment in language design for distributed systems, *Software Practice and Experience* **14** (1984) 957-971.
[9] H.G. Dietz, Finding large-grain parallelism in loops with serial control dependencies, in: *Proc. 17th Internat. Conf. on Parallel Processing*, Vol II (Pen. State Press, 1988) 114-121.
[10] E.W. Dijkstra, Hierarchical ordering of sequential processes, *Acta Inform.* **1** (1971) 115-138.
[11] S.J. Garland and D.C. Luckham, Program schemas, recursion schemas, and formal languages. *J. Comput. System Sci.* **7** (1973) 119-160.
[12] T. Hey, *Experiments in MIMD Parallelism*, Lecture Notes in Computer Science **366** (Springer, Berlin, 1989).
[13] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice Hall, Englewood Cliffs, NJ, 1985).
[14] R.C. Holt, A short introduction to concurrent Euclid, *ACM Sigplan Notices* **17** (1982) 60-79.
[15] Inmos Ltd., *occam Programming Manual* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[16] R. Janicki, *A Construction of Concurrent Systems by means of Sequential Solutions and Concurrency Relations*, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981) 327-334.
[17] R. Janicki, On the design of concurrent systems, in: *Proc. 2nd. Conf. on Distributed Computing Systems*, Paris (IEEE Press, New York, 1981) 455-466.
[18] R. Janicki, *A Method for Developing Concurrent Systems*, Lecture Notes in Computer Science **167** (Springer, Berlin, 1984) 155-166.
[19] R. Janicki, Transforming sequential systems into concurrent systems, *Theoret. Comput. Sci.* **36** (1985) 25-58.
[20] R. Janicki, How to relieve a programmer from synchronization details, in: *Proc. 16th Ann. ACM Computer Science Conf.*, Atlanta, GA (1988).
[21] R. Janicki and T. Müldner, Sequential specifications and concurrent executions of Banach programs, in: *CIPS'88*, Edmonton, Canada (1988).
[22] R. Janicki and T. Müldner, Complete sequential specifications allows for a concurrent execution, in: *ACM 1989 Computer Science Conf.*, Louisville, KY (1989).
[23] R. Janicki and T. Müldner, On algebraic transformations of sequential specifications, in: *AMAST'89*, Iowa City, IA (1989).
[24] R. Janicki and T. Müldner, A simple realization of parallel devices recognizing regularly defined trace languages, in: *ACM 1990 Computer Science, Conf.*, Washington, DC (1990).
[25] R. Janicki, P.E. Lauer, M. Koutny and R. Devillers, Concurrent and maximally concurrent evolution of non-sequential systems, *Theoret. Comput. Sci.* **43** (1985) 213-238.
[26] R. Janicki and P. Lauer, *Specifications and Analysis of Concurrent Systems; The COSY Approach* (Springer, Berlin, 1990) to be published.
[27] M. Jazayeri et al., CSP/80: A language for communicating sequential processes, *IEEE Compcon* (1980) 736-740.

[28] G. Lallement, *Semigroups and Combinatorial Applications* (John Wiley, New York, 1979).
[29] B.W. Lampson and D.D. Redell, Experience with processes and monitors in Mesa, *Comm. ACM* **23** (1980) 105-117.
[30] P.E. Lauer, The COSY approach to distributed computing systems, in: D.A. Duce, ed., *Distributed Systems Programs* (P. Peregrinus, London, 1984) 107-126.
[31] P.E. Lauer and R. Janicki, An introduction to the MACRO COSY notation, in: K. Voss, H.J. Genrich and G. Rozenberg, eds., *Concurrency and Nets* (Springer, Berlin, 1987) 287-314.
[32] C. Lengauer, A methodology for programming concurrency: the formalism, *Sci. Comput. Programm.* **2** (1982) 19-52.
[33] C. Lengauer and E.C.R. Hehner, A methodology for programming concurrency: an informal approach, *Sci. Comput. Programm*, **2** (1982) 1-18.
[34] A. Mazurkiewicz, Recursive algorithms and formal languages, *Bull. Acad. Polon. Sci. Ser. Math. Astronom. Phys.* **20** (1972) 799-803.
[35] A. Mazurkiewicz, Concurrent program schema and their interpretations, *Report DAIMI-PB-78*, Aarhus University, 1977.
[36] A. Mazurkiewicz, *Trace Theory*, Lecture Notes in Computer Science **255** (Springer, Berlin, 1986) 297-324.
[37] T. Müldner and P. Prószyński, Sequential specification of the dining philosophers problem, in: *APICS'89 Computer Science Conf.*, Fredericton, NB (1989); also published as a Technical Report, Acadia University.
[38] A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems* (Sequent Computer Systems Inc., 1986).
[39] W. Reisig, *Petri Nets* (Springer, Berlin, 1985).
[40] *Reference Manual for the Ada Programming Language*, United States Department of Defence (U.S. Government Printing Office, Washington, DC 20402, 1982).
[41] K. Smith and W.F. Appelbe, FAT—An interactive Fortran parallellizing tool, in: *Proc. 17th Internat. Conf. on Parallel Processing*, Vol II (Pen. State Press, 1988) 58-62.
[42] M.W. Shields, *Adequate Path Expressions*, Lecture Notes in Computer Science **70** (Springer, Berlin, 1979) 249-265.
[43] M.W. Shields, Concurrent machines, *The Computer Journal* **25** (1985) 449-466.
[44] J. Welsh and D.W. Bustard, Pascal-Plus—another language for modular multiprogramming, *Software Practice and Experience* **9** (1979) 947-957.
[45] N. Wirth, Modula—a language for modular multiprogramming, *Software Practice and Experience* **7** (1977) 3-35.