PROGRAMMING AND VERIFYING ASYNCHRONOUS SYSTEMS

J. Y. COTRONIS

Computing Laboratory

July 1982

Ph.D. Thesis

University of Newcastle-upon-Tyne

AKNOWLEDGEMENTS

I would like, first of all to express my thanks to my supervisor Dr. Peter Lauer for his continuous cooperation during the years I have been a student and later a member of his research project. Peter's comments on early drafts of my thesis were valuable and in a great extend determined the contents of the thesis. In particular Peter suggested the four criteria for a good macro notation. His work in [L79] provided the guidelines for developing syntax rules for macro programs uniform with syntax rules for basic programs. Peter also suggested the elimination of the box " " in replicators and the restriction of the "," as far as possible in regular expressions denoting choice. Peter also suggested possible theorems to be proved about macro programs.

I would like to thank Brian Hamshere for his patience in reading through an early version of my thesis suggesting improvements and corrections. I would also like to thank old members of the Asynchronous Systems Project, Dr. Mike Shields and Dr. Eike Best for providing the basis for the research in this thesis.

Finally, I would like to express my warmest thanks to my parents and to Lily who many a times, regretably, took second place to this thesis, for their encouragement and support during the past years.

ABSTRACT

The basic COSY (COncurrent SYstems) notation [LSB79b] is briefly presented. Programs in this notation abstractly specify the synchronic aspects of concurrent systems and are possessed of behavioural semantics, which are capable of expressing concurrency and which also provide a firm mathematical foundation for verifying properties of systems.

We are mainly concerned with the macro COSY notation [LTS79] which contains macro features for concisely representing and precisely generating by expansion similar regularities of structure of programs in the basic notation. We re-examine and revise all aspects of macro COSY, the design of the notation as a specification language, the formal grammar for producing macro COSY programs, the rules for the expansion of macro elements and of complete macro programs, eliminating serious drawbacks of previous macro COSY notations and grammars.

We characterize the strings generated by the expansion of macro elements and macro programs and we investigate the conditions under which macro elements may generate the same strings as other macro elements.

Finally, we give direct semantics to macro programs following two approaches.

CONTENTS

1 INTRODUCTION

In the past few years there has been an increasing interest in distributed computing systems, that is systems in which there are a number of autonomous but interacting computers co-operating on a common problem. Such systems cover a broad spectrum which includes networks of main-frame computers, systems containing microprocessors, novel forms of highly parallel computer architecture etc. Recent and continuing developments in component technology have initiated new ideas on system design, based on the decomposition of systems into a number of subsystems which when combined in new ways may perform the same general functions as earlier systems but with much greater degree of parallelism and distribution. These new design options have at the same time increased the difficulties for the precise specification, analysis and verification of systems.

The COSY notation [LSB79b, LTS79], the name has been derived from COncurrent SYstems, is a formalism indented to simplify these tasks by abstracting away from all aspects of systems, except those which have to do with synchronization. In the COSY methodology systems are considered as consisting of notionally indivisible actions or events, the occurrences of which may be related to other events in the system. Systems are also assumed to be decomposed into a collection of sequential subsystems each involving a subset of the events of the whole system. Thus the events in the system are left uninterpreted and only the synchronic properties of systems are considered, those which solely concern the ordering of occurrences of these events. That is to say, only properties of a behavioural nature are of interest in the COSY methodology.

System behaviour is abstractly specified in COSY by programs, consisting of operations which correspond to events in the system, together with ordering relationships between their activations, specified in such a way that each relationship determines possible sequences of occurrences of subsets of these operations. These sequences are represented by regular expressions [CH74] which are incorporated in COSY [LC75, LSB79b] and are called path expressions and process expressions or paths and processes respectively, for short. A

single path or process is used for specifying the sequential constraint relating all the operations mentioned in the path or process. A system will be associated with a program, a grammatical type object, consisting of a collection of paths and processes, the "language" of which models a set of permitted or required behaviours. This collection of paths and processes determines a set of vectors of strings of operations. Vectors of strings of operations may be considered as a labelled partial order of operations, modelling a non-sequential behaviour of operation executions, and for this reason they have been called vector firing sequences [SL79]. Vector firing sequences may be shown to have the same modelling power as more conventional models for concurrent behaviour, such as occurrence graphs but have the advantage that may be manipulated in the same manner as strings. The vector firing sequence semantics does not reduce concurrency to arbitrary interleaving [L81] and provides a mathematical environment for the formal definition of system properties and for the analysis of programs, determining whether the system they specify possesses such properties.

The aspects of the notation we have discussed so far constitute the basic COSY notation or basic COSY for short. Programs in this notation involve only paths and processes and are called basic programs.

The COSY notation involves other aspects which were considered essential in a software design environment. Thus, two other notations have been developed, the macro COSY notation and the system COSY notation. The macro COSY notation, macro COSY for short, contains features for the concise representation and precise generation of similar regularities of structure in basic COSY programs. The macro notation was introduced as a matter of convenience for the programmer and as a facility for generalization by allowing the representation and generation of strings of finite but indefinite length. The system COSY notation is equipped with a class-like construct called system, permitting the expression of hierarchy and modularity. Systems allow the specification of levels of abstraction in a design, information hiding and the application of other techniques of structured programming.

In this thesis we are mainly concerned with the macro COSY notation. Since its introduction [L76] macro COSY has been evolved into a flexible and powerful tool for system specification. Three are its main features: the collectivisor, the replicator and the distributor. The collectivisor declares arrays of indexed operations to be used in paths or processes. These arrays may be rectangular but also of other shapes. The replicator is the most general feature for representing and generating a variety of similar regularities of structure in basic programs. These structures include paths and/or processes and regular expressions of paths and processes and their parts. Replicators may generate regularities which either follow each other or are nested within each other. Finally, the distributor may represent and generate some regularities in basic COSY programs. Distributors cannot generate all the regularities which replicators can, but their advantage is that they represent regularities more concisely than replicators. Replicators and distributors are the macro elements of macro COSY and are associated with expansion rules by which they generate basic COSY strings.

Collectivisors, replicators and distributors are used in macro programs. Macro programs do not increase the expressive power of the basic COSY notation as they should expand to basic programs. Macro programs were not given any semantics directly. Their semantics are those of the basic programs they generate.

Although, the need for a macro COSY notation was realized and introduced early in the development of COSY, its development was rightly considered to be an "open-ended" effort. "Open-ended" in the sense that the aim should not be to initially produce a fixed notation, but to permit changes until it is precisely clear what constitutes a "good" macro notation. As a consequence of this approach various macro notations and subnotations have been developed, some being extensions of others or, more commonly, differing in many respects. Some of the differences are for example, that replicators in some notations may generate paths and/or processes whilst in other notations just paths or just processes, that replicators in some notations could be nested inside other replicators and in others not, etc.

Besides this diversity of notations there is a diversity of formal grammars producing macro COSY programs. The differences in the notations and subnotations are certainly reflected in the grammars. In addition, various approaches have been adopted in defining the syntax of macro programs and in particular the syntax of macro elements. These approaches however are not equivalent, in the sense that their corresponding syntax rules do not produce the same classes of macro programs.

The main problem with most grammars is that they may produce "macro programs" which when expanded do not generate basic programs. This was realized and some meta-restriction rules were imposed on macro programs which were to eliminate these "unwanted programs". However, even these "wide" grammars do not permit some programs we would like to write. Thus, grammars are too "wide" in some aspects and at the same time restrictive in others. The need for a context-free grammar producing exclusively macro programs expanding to basic programs was realized and some close-fitting syntax rules were suggested [L79] but overconstrained the class of valid macro programs.

There are also a number of minor aspects of the macro notation and grammar which need to be improved such as that some symbols are awkward to use, that the syntax of some features of macro programs has never been obtained, and others.

Another aspect of the macro notation are the expansion rules for replicators, distributors and of complete macro programs. Whilst the expansion of replicators was formally defined that of distributors was not formally defined directly.

The objectives of this thesis are to re-examine and revise all aspects of the macro notation, its design as a specification language, the formal syntax of macro programs, the expansion rules of macro elements and of complete macro programs, alleviating or eliminating altogether the drawbacks of other notations and grammars; to characterize the strings generated by the expansion of replicators, distributors and of complete macro programs produced by the formal grammar; to investigate some aspects of programming methodology such as

when replicators and distributors may be replaced by other replicators and distributors expanding to the same string as the former; and finally to give direct vector firing sequence semantics to macro programs rather than indirectly via the basic programs generated by their expansion.

Our guidelines for revising the macro notation and grammar were mainly four:

1. The syntactic well-formedness of a macro program should imply that its expansion is a syntactically well-formed basic program.

2. The notation should allow the generation of a large class of basic programs and their concise representation.

3. The macro grammar should include context-free rules and should be uniform with the grammar of basic COSY.

4. The reading of macro programs should be possible without formal expansion.

In the design of the notation, changes of symbols and of the forms of the collectivisors, replicators and distributors are suggested improving the readability of these constructs and of the macro programs as a whole. Some restrictions imposed on what replicators may generate ensure the readability of unexpanded macro programs. But the new replicators may generate strings which could not be generated by a single replicator in previous notations. Distributors are extended to generate more strings more economically than replicators. Two new types of replicators are added generating strings which could not be generated by replicators in previous notations. It is precisely specified where distributors and each type of replicators should appear in macro programs.

The new context-free syntax rules for macro programs combine some of the syntax rules of previous grammars, modified to be consistent with changes in the design of the notation. For the main features of the notation though, that is collectivisors, replicators and distributors

new syntax rules are introduced. Particular attention is given to the problem of obtaining a grammar uniform with the grammar for basic COSY. This is achieved by expressing the new grammar as an extension of the basic COSY grammar and by expressing the syntax of the features of macro programs in a style similar to that of basic COSY.

The expansion rules for replicators are modified to deal with their new form and the expansion of distributors is directly defined. The expansion of replicators and distributors is also characterized. The expansion of complete macro programs is formally defined and it is proven that programs permitted by the new grammar generate syntactically well-formed basic programs. Thus, the suggested grammar is not too wide and no meta-restriction rules are needed to eliminate any "unwanted" programs, that is programs not generating basic programs. The conditions under which replicators and distributors may be replaced by other replicators or distributors are also examined.

Finally, we give direct vector firing sequence semantics to macro programs and we show that the vector firing sequences of macro programs are the same as the vector firing sequences of the basic programs generated by their expansion.

The rest of the thesis is structured as follows: In chapter 2 the basic COSY notation is briefly presented, chapter 3 deals with the syntax and expansion of macro COSY programs, chapter 4 deals mainly with the semantics of macro COSY programs, and chapter 5 contains the conclusions of the thesis. The contents of the main chapters 2, 3 and 4 in more detail are as follows:

Chapter 2 deals with the syntax and semantics of basic COSY programs and briefly with the nature of analysis and verification in COSY. In section 2.1 the syntax of basic programs is given. Section 2.2 gives the semantics of a single path by associating with it a set of strings of operations involved in the path. The elements of this infinite set may be obtained from a set, the set of cycles of a path. Section 2.3 gives the semantics of basic programs consisting exclusively of paths by means of sets of vectors of strings of operations involved in the programs, representing the behaviour of these

programs. Each of the components of these vectors relates to a path in the basic program and must be a possible total order of the operations in that path. Furthermore, all components must agree on the number and order of the activation of operations they share. Besides the usual definition of vector firing sequences, an alternative definition is given which we shall use in chapter 4 where direct semantics are given to macro programs. In section 2.4 the semantics of a general basic program involving paths and processes are given by two methods. The first is the usual method found in the literature for COSY [LS81] and consists of transforming path-process programs into programs involving paths only, the vector firing sequences of which define the behaviour of the original program. The second method obtains vector firing sequences directly from the general basic programs without any intermediate transformation. It is shown that both methods are equivalent in the sense that they produce the same set of vectors. Finally, in section 2.5 a brief account is given on the nature of analysis and verification in COSY.

Chapter 3 is concerned with the syntax and expansion of macro programs. In section 3.1 we review most of macro COSY notations and subnotations in detail, focussing our attention on their formal grammars and discussing the implication of design choices and their drawbacks. In the subsections of 3.1 we examine major syntactic entities of macro programs. In section 3.2 we set the criteria for a "good" macro notation, we revise the macro notation and define the syntax of macro programs. A number of changes, modifications and extensions are introduced. The most important of these are applied to replicators and distributors. In section 3.3 we define the expansion of replicators and distributors and of complete macro programs. We prove four theorems which characterize the strings obtained by the expansion of replicators and distributors. We also show under which conditions replicators may replace distributors and vice-versa, and replicators may be replaced by other replicators. Finally, we formally define the expansion of complete macro programs and prove that they yield well-formed basic programs. In section 3.4 we evaluate the new notation and grammar and discuss certain extensions we could incorporate.

Chapter 4 is concerned with obtaining the vector firing sequences of basic programs generated from macro programs directly from the macro programs themselves. We reduce this task to the task of finding the cycle sets of the paths of such basic programs directly from the macro programs. Two approaches are followed. According to the first which is presented in section 4.1, the cycle sets are constructed by finding the cycle sets of expanded parts of macro programs which are then combined together. We applied the first approach to macro programs produced by the grammar of section 3.2 and to macro programs produced by a restrictive grammar introduced in section 4.1.2. According to the second approach which is presented in section 4.2, the cycle sets of basic paths may be found by constructing macro cycle objects from macro programs representing cycle sets concisely, which may be expanded to generate cycle sets, in the same way macro programs are expanded to generate basic programs. The second approach is applied to macro programs which are produced by the grammar of section 4.2.1, obtained by constraining the grammar in section 4.1.2.

2 THE BASIC COSY NOTATION

In the COSY methodology systems are considered as consisting of notionally indivisible actions or events, the occurrences of which may be related to occurrences of other events in the system. Thus the events in the system are left uninterpreted and only the synchronic properties of systems are considered, those which solely concern the ordering of occurrences of these events. That is to say only properties of a behavioural nature are of interest in the COSY methodology.

The COSY notation is a formalism which may be used to describe concurrent and distributed systems in their synchronic properties. The notation used was basically the path notation due to Campbell and Habermann [CH74] which was designed so that one could state the proper coordination of concurrent processes as the permissible order of execution of operations on shared system objects as part of the object definition. The idea behind the Campbell-Habermann path concept was put into a more abstract form, the path and process expressions of Lauer and Campbell [LC75], or paths and processes for short. Later this notation was named the basic COSY notation [LSB79b]. System behaviour may be specified by programs consisting of collections of paths and processes, that is basic programs. Paths and processes are essentially regular grammars represented by regular expressions. Just as a single regular expression determines a set of strings, each of which may be considered as a labelled total order modelling a sequence of execution of operations which label it, so may a basic program, a collection of regular expressions, determine a set of vectors of strings, where each vector may be considered as a labelled partial order, modelling a non-sequential behaviour of operation executions.

In the next section 2.1 the syntax of basic programs is given. In section 2.2 the semantics of a single path P are given, by means of the possible sequences of the activation of operations involved in its regular expression, the set of its firing sequences denoted by FS(P). In section 2.3 the semantics of a basic program consisting only of paths is defined by means of a mapping which associates with each program R the set of its vector firing sequences denoted by VFS(R) consisting of vectors of strings of operations in R. In section 2.4 the semantics of a general

basic program consisting of paths and processes are given in two ways: According to the first, a transformation Path is defined which transforms a general basic program R into a basic program denoted by Path(R) consisting of just paths. Then the semantics of R are defined in terms of the vector firing sequences of Path(R) denoted by VFS(Path(R)). According to the second way the same semantics are obtained without R having to undergo any transformation. In section 2.5 the nature of analysis and verification of COSY programs is outlined.

2.1 THE SYNTAX OF BASIC COSY

A basic COSY program is a string derived from the production rules given below. The following meta-language conventions have been used in the syntax rules: The symbols "=", "{", "}", "/", "*", "+", "@" have been used as meta-symbols. The symbol "=" denotes production of its left hand side to strings on its right hand side. The braces "{ }" are used to group items together, "/" indicates alternate productions, "{item}*" indicates production of "item" zero or more times, "{item}+" production of "item" one or more times. The notation

{item1 @ item2}+

is used as a shorthand for

item1 {item2 item1}*

In the syntax rules for basic COSY programs "item2" may be one of the terminal symbols ";" and ",". Non-underlined lower case words, except single lower case letters and digits, are non-terminal symbols, and all other symbols like ";", ",", "(", ")", "*", undelined lower case words and single lower case letters and digits are terminal symbols. We shall additionally use the following convention: in right parts of production rules the catenation of terminals and non-terminals has precedence over alternation. Thus A B/C means either A B or C. When necessary we use "{ }" to override the normal precedence. Thus A {B/C} means either A B or A C.

The syntax of a basic COSY program is given by the following rules:

BN1. basicprogram = program programbody endprogram
BN2. programbody = {path/process}+
BN3. path = path (sequence)* end
BN4. process = process (sequence)* end
BN5. sequence = {orelement @;}+
BN6. orelement = {starelement @,}+
BN7. starelement = element/element*
BN8. element = operation/(sequence)
BN9. operation = lc-letter{lc-letter/digit/_}*
BN10. lc-letter = a/b/.../z
BN11. digit = 0/1/.../9

In the regular expressions produced by the non-terminal "sequence" the symbols ";" and "," denote sequentialization and arbitrary choice respectively; the symbol "*" is the Kleene star.

All the regular expressions in paths and processes are considered to be cyclic in the sense that constituent operations may be executed repeatedly subject to the constraints of sequentialization and arbitrary choice. For this reason the outermost star and parentheses are usually omitted, their presence being implicit.

2.2 THE SEMANTICS OF A BASIC PATH

The semantics of a basic path P are given in terms of its set of firing sequences denoted by FS(P). The infinite set FS(P) may be constructed from a set consisting of the cycles of P. Let us define the function "Cyc" by which the cycles of a basic path P may be constructed. The function "Cyc" will apply to syntactic entities of basic paths, that is to say substrings produced by non-terminals. Syntactic entities of paths will be denoted by syntactic variables. A path P will be represented by

path (SEQ)* end

where SEQ denotes a sequence, which may be represented by

OREL1;...;ORELn

where ORELi for i=1,...,n denote orelements. An orelement may be represented by

STAREL1,...,STARELn

where STARELi for i=1,...,n denote starelements. A starelement may be represented by

ELEM* or ELEM

where ELEM denotes an element which may be represented by

(SEQ)

when it is produced by the second option of the syntax rule for element BN8, or by

OP

when produced by the first option. The function "Cyc" is defined as follows:

Cyc(e)=cases e:

1. path (SEQ)* end → Cyc(SEQ)
2. OREL1;...;ORELn → Cyc(OREL1) ⊙...⊙ Cyc(ORELn)
3. STAREL1,...,STARELn → Cyc(STAREL1) U...U Cyc(STARELn)
4. ELEM* → Cyc(ELEM)*
5. (SEQ) → Cyc(SEQ)
6. OP → {OP}

The function "Cyc" is defined in terms of a "case-function". A function f defined by cases

f(e)=cases e:

1. c1 →fc1
2. c2 →fc2
   ...
n. cn →fcn

in which c1,c2,...,cn are the valid forms which expression e may take, has the following semantics:

if e is of form c1 then f(e)=f(c1) converts into fc1 else
if e is of form c2 then f(e)=f(c2) converts into fc2 else
   ...
if e is of form cn then f(e)=f(cn) converts into fcn

In the definition of "Cyc" the symbol "U" denotes the set-union operator and the symbol " " the concatenation of sets of strings

operator. The operation

$$X \circ Y$$

where X, Y aresets of strings is defined as:

$$X \circ Y = \{x.y \mid x \in X, y \in Y\}$$

where "." denotes string concatenation and " $\in$ " element of a set.

In the definition of "Cyc" a starred set X* indicates the set obtained by concatenation of zero or more times of the set X. Formally X* is defined by

$$X^* = X^0 \cup X^1 \cup X^2 \cup \ldots$$

where X is a set of strings and Xi is defined recursively by

$$Xi = Xi\text{-}1 \circ X$$
$$X^0 = \{\lambda\}$$

where "λ" denotes the empty string.

From the set Cyc(P) we may construct the set of firing sequences of P denoted by FS(P) as follows:

$$FS(P) = Pref(Cyc(P)^*)$$

where Pref(X) is defined as

$$Pref(X) = \{x \mid x.y \in X, \text{ for some } y\}$$

where X is a set of strings.

The set FS(P) is the set of sequences of operation executions permitted by the path P.

2.3 THE SEMANTICS OF PATH-PROGRAMS

As already mentioned, to model the non-sequential behaviour of a basic program R consisting of paths Pl,...,Pn partial orders of occurrences of operations will be constructed which are specified by vectors of strings. An n-vector $\underline{x}$

$$\underline{x}=(x1,\ldots,xn)$$

is a possible behaviour of R if each xi for $1\leq i\leq n$ is a possible firing sequence of Pi for i=1,...,n and furthermore, if the xi's agree on the number and the order of activation of operations they share.

To formally define the set of possible behaviours or histories of R, vectors of strings are introduced together with a composition operation on them. Let Sl,...,Sn be a family of sets of strings and let

$$\overset{n}{\underset{i=1}{\times}} Si^*=Sl^* \times\ldots\times Sn^*=\{(sl,\ldots,sn)\mid \text{for all } i,\ si \in Si^*\}$$

where "×" denotes the cross product operator. If the vectors $\underline{x}$ and $\underline{y}$ belong to the above set then their composition $\underline{x}\circ\underline{y}$ is defined as

$$\underline{x}\circ\underline{y}=(x1,\ldots,xn)\circ(y1,\ldots,yn)=(x1.y1,\ldots,xn.yn)$$

where "o" denotes the vector concatenation operation and the "." denotes string concatenation operator.

To each program R consisting exclusively of paths

$$R=P1\ldots Pn$$

we associate its set of operations Ops(R) defined by

$$Ops(R)=Ops(P1)U\ldots U\ Ops(Pn)$$

and its set of vector operations Vops(R) defined as follows:

For each operation "a" in R we construct an n-vector $\underline{a}$. The

i'th component of this vector for $1 \leq i \leq n$ denoted by $[\underline{a}]i$ is given by

$$[\underline{a}]i = \begin{cases} a \text{ if } a \in Ops(Pi) \\ \lambda \text{ otherwise} \end{cases}$$

where "λ" denotes the null string.

The set of vector operations of R, Vops(R) is then defined as

$$Vops(R) = \{\underline{a} \mid a \in Ops(R)\}$$

Let us define Vops(R)* to be the submonoid of

$$\overset{n}{\underset{i=1}{\times}} Ops(Pi)^*$$

generated by Vops(R) and $\underline{\lambda}=(\lambda,\ldots,\lambda)$ under the vector composition operation. The set of all possible behaviours or histories of R, the vector firing sequences of R, denoted by VFS(R) is defined by:

$$VFS(R) = \left(\overset{n}{\underset{i=1}{\times}} FS(Pi)\right) \cap Vops(R)^*$$

The set

$$\overset{n}{\underset{i=1}{\times}} FS(Pi)$$

in the definition of VFS(R) guarantees that each string component of a history $\underline{x} \in VFS(R)$ is a firing sequence of the corresponding path and the set Vops(R)* guarantees that all these firing sequences agree on the number and order of activations of the operations they share.

By the construction of VFS(R) every element $\underline{x}$ of it represents everything that has happened in some possible period of activity of R. We may write $\underline{x}$ as a composition of vector operations $\underline{a1},\ldots,\underline{am}$ of Vops(R) as in (V1)

(V1) $x = a1 \circ \ldots \circ am$

If for some operations "ak" and "al" for $1 \leq k, 1 \leq m$ and $k \neq 1$, $[ak]i \neq e$ implies $[al]i = e$ for $i = 1, \ldots, n$ then the composition $ak \circ al$ is the same as $al \circ ak$. Such operations are said to be independent and we write ind(ak,al). If furthermore $l = k+1$ that is ak and al are neighbouring vectors in (V1), as in (V2)

(V2) $x = a1 \circ \ldots ak \circ al \circ \ldots am$

then x may also be written as (V3)

(V3) $x = a1 \circ \ldots \circ al \circ ak \circ \ldots \circ am$

The commutativity of vector operations in a vector firing sequence is interpreted to mean that the operations corresponding to these vector operations may execute concurrently. We say that

two operations "a" and "b" are concurrent at a history x and we write

a co b at x

if ind(a,b) and $x \circ a$, $x \circ b$ $\in$ VFS(R).

This definition implies that only independent operations may execute concurrently. However, independent operations may not always be executable concurrently or may never execute concurrently at all. Let us consider the basic program (R1)

(R1)

```
program
  path a ; b end
  path b ; d end
endprogram
```

Although ind(a,d) and operation "a" may be executed initially, the operation "d" cannot be executed. However, whenever the operation "d"

can be executed so can the operation "a" and since they are independent they may be executed concurrently. For example, after the history $\underline{a} \circ \underline{b}$ the operations "a" and "d" may be concurrently executed.

It may happen that two independent operations cannot be executed concurrently at all. This occurs when there is not a history after which both operations may execute. Consider for example the basic program (R2)

```
(R2)
program
  path a ; b ; d end
  path b ; c ; d end
endprogram
```

Although operations a and c are independent that is $\underline{ind}(a,c)$ there is not an $\underline{x} \in VFS(R2)$ such that

$$\underline{x} \circ \underline{a} \ , \ \underline{x} \circ \underline{c} \in VFS(R2)$$

as the second path specifies that operation c occurs after b and before d, and operations a, b and d are sequentialized in the first path.

For the construction of the vector firing sequences of a basic program R, the following sets need to be constructed directly from R:

1. the cycle sets of all paths in R, and

2. the set of the vector operations in R, Vops(R).

There is a modification of this construction by which the latter set is obtained from the sets of cycles of the paths of R and not from the program R. This alternative construction will be useful in the fourth chapter where we construct the vector firing sequences of basic programs generated from macro programs directly from the macro programs themselves. The sets of cycles of a basic program generated from a macro program will be constructed directly from the macro program itself. The set of vector operations however, cannot easily be obtained

directly from the macro program but may be obtained, as we show below from the sets of cycles of paths.

Let s be a string of concatenated symbols s1,...,sn:

s=s1. ... .sn

and denote by (s)i the i'th constituent symbol of s for $1 \leq i \leq n$ and by |s| the length of s. We may now obtain the set Ops(Cyc(P)), the set of operations appearing in the cycle set Cyc(P) of the path P as follows:

Ops(Cyc(P))={a|a=(s)i for s $\in$ Cyc(P)and $1 \leq i \leq |s|$}

The two sets Ops(P) and Ops(Cyc(P)) are the same since all the operations involved in P must appear in at least one string of the cycle set of P, as a single path cannot exclude any of its operations from executing. Having found the operations involved in each path of R we proceed by constructing Ops(R) and Vops(R) as before.

2.4 THE SEMANTICS OF GENERAL BASIC PROGRAMS

In general, a basic program R is a string of the form

R=P1...Pn Q1...Qm

where Pj for j=1,...,n and Qi for i=1,...,m denote paths and processes respectively. Although paths and processes may be intermixed in a basic program, in the above expressions for convenience, we assumed that all paths are collected before processes.

In the COSY literature e.g [LS81] the semantics of a basic program involving processes is given by means of the vector firing sequences of an equivalent basic program R' involving just paths. The conversion of R into R' is denoted by Path(R) and is obtained by the following rule:

(Path Conversion Rule)

1. For every a $\in$ Ops(R) construct a set

   $Ia=\{i \mid a \in Ops(Qi) \text{ for } 1 \leq i \leq m\}$

   and, if the cardinality of the set Ia denoted by |Ia| is greater than zero, say $l=|Ia|>0$ then

   - replace the operation "a" in each path it occurs $\overset{in}{\vee}$ by the element

     (a&i1,...,a&il)

     where ik $\in$ Ia for k=1,...,l

   - replace the operation "a" in processes Qik by a&ik for all ik $\in$ Ia.

2. Replace all occurences of "process" by "path".

Then the semantics of R are given by means of VFS(Path(R)) and are obtained as defined in the previous section.

Besides some differences of formulation between the way the path conversion rule $\overset{is}{\vee}$ expressed in [LS81] and above, there is one another important difference. The rule in [LS81] specifies that an operation "a" occuring in processes is replaced in each path it occurs $\overset{in}{\vee}$ by the orelement OREL

OREL a&i1,...,a&il

When however, the operation "a" is starred it should not be replaced by OREL but by the element

(a&i1,...,a&il)

In the above rule we generalized this replacement to avoid considering

cases and we treat all the operations in the same way simplifying the conversion rule.

The relation between the basic path program Path(R) and its vector firing sequences VFS(Path(R)) is already defined in 2.3. Here we have to relate the behaviour of Path(R) with that of R since R is the program the semantics of which we seek. Let us first introduce some terminology. We shall call the operations of the form

a&ik for ik € Ia

the descendent operations of a. The behaviours of Path(R) and R are related as follows:

If an operation "op" may be activated in Path(R) then

1. If op is not a descendent operation of any operation in R then it may also be activated in R.

2. If op is a descendent of some operation "a" in R of the form "a&j" for j € Ia, then the operation "a" may be activated in R and out of all processes requiring its activation to progress, process Qj will be granted it.

The set of histories VFS(Path(R)) may be obtained without R having to undergo any conversion, that is it may be obtained directly from R. In the method which follows the definition of firing sequences of paths and vector operations of programs defined in the previous section are modified and firing sequences for processes are defined. A program R is considered to be of the form

R=S1...Sn+m

where Si for i=1,...,n are paths and Sj for j=n+1,...,m processes. Let us denote the set of histories obtained by this method by MVFS(R) standing for modified vector firing sequences. Let us define the set of the modified firing sequences of paths and processes in R by

MFS(Sj)=Pref(MCyc(Sj)*) for j=1,...,n+m

where MCyc(Sj) denotes the modified cycles of Sj for j=1,...,n+m. We shall distinguish two cases for the construction of MCyc(Sj) depending on whether Sj for j=1,...,n+m is a path or process. The function MCyc(Sj) is defined by

$$MCyc(Sj)=\begin{cases} path\text{-}Cyc(Pj) & \text{if } j=1,\ldots,n \\ proc\text{-}Cyc(Qj-n,j-n) & \text{if } j=n+1,\ldots,n+m \end{cases}$$

where path-Cyc(Pj) for j=1,...,n denote the cycle sets of paths Pj for j=1,...,n and proc-Cyc(Qj-n,j-n) for j=n+1,...,n+m the cycle sets of processes Qj-n for j=n+1,...,m.

The function "path-Cyc" will be applied to the same syntactic entities as the function "Cyc" both yielding the same results except when applied to an operation OP belonging to processes. In this case "path-Cyc" will yield the set of descendent operations of operation OP. The function path-Cyc(Pj) is defined as follows:

path-Cyc(e)=<u>cases</u> e:

1. <u>path</u> (SEQ)* <u>end</u> → path-Cyc(SEQ)
2. OREL1;...;ORELk → path-Cyc(OREL1) ⊙...⊙ path-Cyc(ORELk)
3. STAREL1,...,STARELk → path-Cyc(STAREL1) U...U path-Cyc(STARELk)
4. ELEM* → path-Cyc(ELEM)*
5. OP → $\begin{cases} \{OP\} & \text{if } |Iop|=0 \\ \{OP\&i \mid i \in Iop\} & \text{if } |Iop|>0 \end{cases}$
6. (SEQ) → path-Cyc(SEQ)

The function "proc-Cyc" will have two arguments. The first are syntactic entities in processes, the same as in paths. The second is the integer indexing processes and remains unaltered for a given process. The effect of "proc-Cyc" on the first argument is the same as that of "Cyc" with the exception of the case when the syntactic entity is an operation OP. In this case "proc-Cyc" yields one of the descendent operations of OP namely OP&j where j is second argument of "proc-Cyc". The function proc-Cyc(Qj,j) is defined as follows:

proc-Cyc(e,j)=cases e:

1. process (SEQ)* end → proc-Cyc(SEQ,j)
2. ORELl;...;ORELk → proc-Cyc(ORELl,j) o...o proc-Cyc(ORELk,j)
3. STARELl,...,STARELk → proc-Cyc(STARELl,j) U...U proc-Cyc(STARELk,j)
4. ELEM* → proc-Cyc(ELEM,j)*
5. OP → {OP&j}
6. (SEQ) → proc-Cyc(SEQ,j)

Let us define the sets of operations occurring exclusively in paths denoted by Pops(R) and operations occurring in processes denoted by Qops(R) of a program R by

$$Pops(R)=\{a \mid a \in Ops(R), |Ia|=0\}$$
$$Qops(R)=\{a\&i \mid a \in Ops(R), i \in Ia\}$$

and the set of all operations in R denoted by Mops(R) by

$$Mops(R)=Pops(R) \cup Qops(R)$$

Let us now define two sets of vector operations of operations in Pops(R) and Qops(R) denoted by VPops(R) and VQops(R) respectively. The set Vops(R) is defined by

$$VPops(R)=\{\underline{a} \mid a \in Pops(R)\}$$

where $\underline{a}$ is an (n+m)-vector the j'th component of which, for $1 \leq j \leq n+m$, denoted by $[\underline{a}]j$, is given by:

$$[\underline{a}]j=\begin{cases} a \text{ if } 1 \leq j \leq n \text{ and } a \in Ops(Pj) \\ \lambda \text{ otherwise} \end{cases}$$

The set VQops(R) is defined by:

$$VQops(R)=\{\underline{a\&i} \mid a\&i \in Qops(R), i \in Ia\}$$

where $\underline{a\&i}$ is an (n+m)-vector the j'th component of which, for $1 \leq j \leq n+m$, denoted by $[\underline{a\&i}]j$, is given by:

$$[\underline{a\&i}]j=\begin{cases} a\&i \text{ if } 1\leq j\leq n \text{ and } a \in Ops(Pj) \text{ or } j=i+n \\ \lambda \text{ otherwise} \end{cases}$$

Let us finally denote the set of the vector operations in R by MVops(R) and define it by

$$MVops(R)=VPops(R)\cup VQops(R)$$

We may now define the set of histories of R denoted by MVFS(R) by

$$MVFS(R)=(\overset{n+m}{\underset{j=1}{\times}} MFS(Sj))\cap MVops(R)^*$$

Having constructed MVFS(R) we need to relate its elements with execution of operations in R. Let us introduce some terminology first. We shall call the vector operations in the form $\underline{a\&j}$ the $\underline{\text{descendent vector operations}}$ of an operation "a".

The relation between MVFS(R) and R is as follows:

If a history $\underline{x} \in$ MVFS(R) may be continued by the vector operation "$\underline{op}$" then

1. if $\underline{op}$ is not a descendent vector operation of R then operation "op" in R may be activated.

2. If "$\underline{op}$" is a descendent vector operation of operation "a" in R of the form $\underline{a\&j}$ then operation "a" may be activated in R and out of all processes requiring the activation of "a" to progress, process Qj will be granted it.

We next prove that the set VFS(Path(R)) is the same as MVFS(R). The symbol "√√√" will indicate "end of proof".

THEOREM 2.1:

For a basic program R of the form

$$R=P1\ldots Pn\ Q1\ldots Qm$$

where Pj for j=1,...,n are paths and Qi for i=1,...,m are processes,

MVFS(R)=VFS(Path(R))

Proof:

We need to prove that

1. $\overset{n+m}{\underset{j=1}{\times}}(MFS(Sj))=\overset{n+m}{\underset{j=1}{\times}}(FS(P'j))$

where P'j represents the j'th path in Path(R) for j=1,...,n+m, and that

2. MVops(R)*=Vops(Path(R))*

Proof of 1. It suffices to prove that

MFS(Sj)=FS(P'j) for j=1,...,n+m

We will distinguish two cases: (a) when Sj is a path and (b) when it is a process.

(a) Sj is a path, that is j=1,...,n.
Since

MFS(Sj)=Pref(path-Cyc(Sj)*) and
FS(P'j)=Pref(Cyc(P'j)*)

we have to prove that

path-Cyc(Sj)=Cyc(P'j)

The function "path-Cyc" is applied to the same syntactic entities as "Cyc". Furthermore, their definitions are exactly the same except in the case in which the syntactic entity is an operation. When an operation "op" does not appear in processes then path-Cyc(op)={op}. The operation "op" belongs to Ops(P'j) since it has not been replaced in Pj

and Cyc(op)={op}.

When an operation "op" does belong to processes

path-Cyc(op)={op&i|i ∈ Iop}

The operation "op" in path Pj is replaced by (op&il,...,op&il) where l=|Iop| and ik ∈ Iop for $1 \leq k \leq l$. According to the definition of "Cyc"

Cyc((op&il,...,op&il))=Cyc(op&il,...,op&il)=
Cyc(op&il)U...U Cyc(op&il)=
{op&il} U... U{op&il}={op&i|i ∈ Iop}

Therefore, MFS(Sj)=FS(P'j) for j=1,...,n.

(b) Sj is a process, i.e. j=n+1,...,n+m.
Since

MFS(Sj)=Pref(proc-Cyc(Sj)*)
FS(P'j)=Pref(Cyc(P'j)*)

we have to prove that

proc-Cyc(Sj)=Cyc(P'j) for j=n+1,...,m

A process Qj in R for j=1,...,m of the form

Qj=process (SEQ)* end

is converted into the path P'j+n of the form

P'j+n=path (SEQ')* end

in which SEQ' is obtained from SEQ by replacing each operation in SEQ by its name suffixed by "&j". Therefore,

Cyc(P'j+n)=Cyc(path (SEQ')* end)

is the same as

Cyc(path (SEQ)* end)

after replacing each operation name in all the strings in the above set by its operation name suffixed by "&j". This however, is the set produced by proc-Cyc(Qj,j). Therefore,

MFS(Sj)=FS(P'j) for j=n+1,..,n+m

Proof of 2. It suffices to prove that

MVops(R)=Vops(Path(R))

First, we observe that

Ops(Path(R))=Mops(R)=Pops(R)U Qops(R)

We shall show that for any a ∈ Ops(Path(R)) the vector operation $\underline{a}$ ∈ Vops(R) and $\underline{a}$ ∈ MVops(R) are the same. If R consists of n paths and m processes in either case $\underline{a}$ will be an (n+m)-vector.

We shall distinguish two cases: (a) operation a occurs only in paths and (b) operation a occurs in processes.

(a) When operation "a" occurs only in paths the j'th component of $\underline{a}$ ∈ Vops(Path(R)) denoted by $[\underline{a}]j$ for j=1,...,n+m is given by

$$[\underline{a}]j=\begin{cases} a \text{ if } a \in Ops(P'j) \\ \lambda \text{ otherwise} \end{cases}$$

which is the same as $\underline{a}$ ∈ MVops(R) defined by

$$[\underline{a}]j=\begin{cases} a \text{ if } 1\leq j\leq n \text{ and } a \in Ops(Sj) \\ \lambda \text{ otherwise} \end{cases}$$

since for $1\leq j\leq n$ Sj is Pj.

(b)when an operation "a" belongs to processes, it is eliminated in Path(R) and descendent operations of the form "a&i" are introduced where i ∈ Ia. The j'th component of the vector operations a&i ∈ Vops(Path(R)) denoted by [a&i]j for j=1,...,n+m is defined by:

$$[a\&i]j = \begin{cases} a\&i \text{ if } a\&i \in Ops(P'j) \\ \lambda \text{ otherwise} \end{cases}$$

Since "a&i" for i ∈ Ia appears in paths P'j of Path(R) corresponding to paths Pj of R and in the P'i+n path of Path(R) corresponding to the process Qi of R, the vector operation a&i above is the same as a&i ∈ MVops(R) defined as

$$[a\&i]j = \begin{cases} a\&i \text{ if } 1 \leq j \leq n \text{ and } a \in Ops(Sj) \text{ or } j=i+n \\ \lambda \text{ otherwise} \end{cases}$$

Therefore, VFS(Path(R))=MVFS(R).√√√

We may just add, for reasons of completeness, that basic programs were at first [LC75] given formal semantics in terms of Petri-nets [P73]. A construction was defined which associated any path-process program with a marked, labelled transition net which was intended to express its "meaning". The net semantics of [LC75] have since been modified [LSB79a] but the central idea remained the same. Each individual path or process, being essentially a regular expression, is associated with a labelled state machine. Putting paths and processes together into a program corresponds to a composition of their associated state machines. The distinction between paths and processes is expressed formally in the nature of the composition in each case.

The current net semantics are based on a composition rule which takes two marked labelled nets N1 and N2 and produces a marked labelled net N1 ⊕ N2 by the identification of transitions with the same label.

2.5 THE NATURE OF ANALYSIS IN COSY

As we have mentioned, a basic COSY program describes a system by specifying partial orders on the execution of its operations and

therefore, the only properties of interest are behavioural in nature.

The formal model of behaviour, the vector firing sequences of path-programs permit us to speak formally of dynamic properties of a system specified by a path-program R. Properties of R may be expressed in terms of its corresponding vector firing sequences VFS(R). Such properties fall into two classes, the general and the specific properties.

The general properties are those which apply to any program, properties such as absence of deadlock or starvation, which may be defined in terms of uninterpreted operations. We say that a path-program R is deadlock-free if and only if

for every $\underline{x} \in VFS(R)$ there exists an $a \in Ops(R): \underline{x} \circ \underline{a} \in VFS(R)$

that is if and only if every history $\underline{x}$ may be continued. We say that

a program R is adequate if and only if

for every $\underline{x} \in VFS(R)$ and for every operation $a \in Ops(R)$
there exists a $\underline{y} \in Vops(R)^*$: $\underline{x} \circ \underline{y} \circ \underline{a} \in VFS(R)$

that is, if and only if every history of R may be continued activating eventually every operation in R. Adequacy is a property akin to absence of partial system deadlock.

The specific properties involve the interpretation of a COSY program as a description of an actual system. The operations of a COSY program are interpreted as actions of a system and the behaviour of the program as the behaviour of the system.

Considerable work has been done concerning the general properties of programs and in particular relating to adequacy [SL78, S79, LS80] and a number of general theorems have been obtained [S79]. For simple comma-free path programs there is a complete characterization of adequacy. Other theorems have been obtained which permit certain program transformations which preserve adequacy.

As far as specific properties of programs are concerned, various programs have been shown to satisfy some design requirements. The most involved of these is the parallel resource releasing mechanism [SL80].

In this short chapter we gave the syntax and the semantics of basic COSY programs, and we briefly outlined the nature of analysis and verification in COSY. The rest of the thesis deals with the macro notation. The next chapter deals with the syntax and expansion of macro programs and chapter four with their semantics.

3 THE MACRO COSY NOTATION

Often in a basic COSY program we find regularities of structure forming various structures like collections of paths and/or processes, sequences, orelements, starelements and elements. For example, let us consider the basic paths specifying the three free frame buffer [LTS79]:

```
P1 path deposit1 ; remove1 end
   path deposit2 ; remove2 end
   path deposit3 ; remove3 end
```

in which the regularity of structure RS1

```
RS1 path depositi;removei end
```

is repeated three times with "i" taking values 1, 2, 3. The regularity RS1 may be used to obtain a more economical representation of P1 or to generalize it by parameterising the number of repetitions of RS1. For simple regularities, such as that of P1, we may denote a repetition of a number of them implicitly by ellipses. For example P1 may be generalized to specify the n free frame buffer [LTS79] by

```
P2 path deposit1 ; remove1 end
                ...
   path depositn ; removen end
```

where the ellipses denote implicitly (n-2) repetitions of the regularity of structure RS1.

When regularities appear within other regularities each having its own ellipses, the unambiguous characterization of the general pattern intended becomes an impossible task. It is apparent that a mechanism for the concise representation of regularities in basic COSY programs is needed from which these regularities may be generated unambiguously. The function of such a mechanism should be twofold:

1. to use the template of a regularity, such as RS1, to make copies of it, differing, if at all, in the names of the operations involved,

and

2. to generate the distinct operation-names in each copy.

A simple way to generate names is to use common or collective names each denoting a collection of operations. Each of these operations may then be represented by a common name subscripted by a set of indices. Now the task of generating names is reduced to the task of generating indices from an index set which may be the set of integers. By convention upper case letters have been used in the identifiers of common names. Following this approach P1 may be rewritten using two common names "DEPOSIT" and "REMOVE" from which one obtains by subscripting the operations "DEPOSIT(i)" and "REMOVE(i)" which correspond to "depositi" and "removei" for i=1,2,3. The basic paths P1 under this transformation become P3:

```
P3 path DEPOSIT(1) ; REMOVE(1) end
   path DEPOSIT(2) ; REMOVE(2) end
   path DEPOSIT(3) ; REMOVE(3) end
```

Strictly speaking P3 is not legal in basic COSY, since subscripted operations are not permitted. For this reason the syntax rule BN9 for the non-terminal "operation" of basic COSY will be replaced by the following three rules:

```
BN9a. operation=simple-op/subscr-op
BN9b. simple-op=lc-letter {lc-letter/digit/_}*
BN9c. subscr-op=uc-letter {uc-letter/digit/_}*({integer @,}+)
```

and the following rules will be added

```
BN12. uc-letter=A/B/.../Z
BN13. integer={digit}+
```

In the above syntax rules we have used the same meta-language conventions as in chapter 2. From now on by a basic COSY program we will mean a string produced by the syntax rules BN1 to BN8, BN9a, BN9b, BN9c, and BN10 to BN12. Programs in this notation should satisfy the following context-sensitive restriction (Brest):

(Brest)

Subscripted operations of the same collective name should have the same number of dimensions.

The semantics of such programs are precisely the same as for programs produced by rules BN1 to BN11 of section 2.1 with the notion of operation extended to cover subscripted operations as well.

The three paths in P3 may be precisely generated by the template RS2

RS2 path DEPOSIT(i);REMOVE(i) end

replicated three times with "i" taking values 1, 2, 3.

This kind of a mechanism was incorporated in the COSY notation forming the macro COSY notation [L76, TL77]. In this notation collective names and their permitted sets of indices are collected by the collectivisors and regularities are concisely represented and precisely generated by replicators and distributors.

Using the macro notation P3 would be represented by FB(3):

FB(3)

C1 array DEPOSIT,REMOVE(3)

P4 [path DEPOSIT(i);REMOVE(i) end i |1,3,1]

in which C1 is the collectivisor declaring the subscripted operations:

DEPOSIT(i) and REMOVE(i) for i=1,2,3

and P4 is the replicator which specifies that the template RS2 is to be replicated and that the values index "i" takes, form a finite arithmetic progression which starts from 1 has upper limit 3 and difference 1, that is it takes the values 1, 2, 3. An n free frame buffer may be specified simply and concisely and generated precisely by generalising FB(3) to FB(n):

```
FB(n)
     C2 array DEPOSIT,REMOVE(n)
     P5 [path DEPOSIT(i);REMOVE(i) end|i|1,n,1]
```

which differs from FB(3) in that the number of operations in each collection of subscripted operations and the upper bound of the value the index takes have been parameterized by the constant n.

Replicators may also be used to represent and generate regularities in sequences in paths and processes. Let us for example consider the basic path P6

```
P6 path DEPOSIT(1);DEPOSIT(2);...;DEPOSIT(n) end
```

which together with FB(n) sequentializes the deposits on the frames of the free frame buffer. In P6 there is a regularity "DEPOSIT(i);" which is repeated (n-1) times with "i" taking values 1,2,...,(n-1). Using the replicator feature of macro COSY the path P6 may be concisely represented avoiding the ellipses by P7

```
P7 path [DEPOSIT(i);|i|1,n-1,1] DEPOSIT(n) end
```

Path P4 may be represented even more concisely by repeating "DEPOSIT(i);" n times and dropping the final ";" after "DEPOSIT(i)". This in macro COSY is specified by P8:

```
P8 path [DEPOSIT(i)@;|i|1,n,1] end
```

in which the "@" is an operator which strips the ";" after the final copy of "DEPOSIT(i);", that is when i=n.

Replicators of the form used in P8 occur so frequently that a shorthand has been introduced, the distributor. The distributor which generates P6 is simply:

```
P9 path ;(DEPOSIT) end
```

assuming that the collective name DEPOSIT has been previously declared

by the collectivisor C2. The distributors do not generate indices explicitly, like the replicators, but generate indices defined by the collectivisors. Although more complex distributors have been used in the notation, which we shall examine in the next section, for each of them there exist replicators which represent and generate the same regularities. The distributors cannot represent and generate all the regularities that replicators can and certainly they cannot represent or generate regularities that replicators cannot. The distributors may only represent and generate some special kind of regularities more economically than replicators. The distributor for example, cannot generate regularities which are nested within each other. In the sequence of the path P10, for example, specifying the stack of size three

P10 path (UP(1);(UP(2);(UP(3);DOWN(3))*;DOWN(2))*;DOWN(1))* end

the starelement "(UP(3);DOWN(3))*" is nested within the starelement "(UP(2);...;DOWN(2))*", which in turn is nested within the starelement "(UP(1);...;DOWN(1))*". To generate this imbrication of regularities another type of replicator has been used. According to it P10 may be generated by:

P11 path [(UP(i)@;[i];DOWN(i))*|1,3,1] end

which may be easily parameterized to specify a stack of size n by

P12 path [(UP(i)@;[i];DOWN(i))*|1,n,1] end

Replicators and distributors do not extend the descriptive power of basic COSY. They merely represent strings of indefinite but finite length of basic COSY concisely. The expansion of the replicators by which they generate the basic regularities they represent, has been defined [LS80] as follows:

If a replicator is of the type T1

T1 [p [i] q |in,fi,inc]

where "p" and "q" are patterns involving the index i and "in", "fi", "inc" are integer expressions, then its expansion is given by Rule1

(Rule1)

```
|empty
|
|     if inc=0 or (fi>in and inc<0) or (in>fi and inc>0)
|
|substitute(p,i,in) [p i q|in+inc,fi,inc] substitute(q,i,in)
|
|     otherwise
```

where "substitute(pattern,index,value)" indicates the string obtained from "pattern" by substituting every occurrence of the "index" by the integer value "value". If the replicator involves the "@" , thus being of the form T2

```
T2 [ p@s1  i  q@s2 |in,fi,inc]
```

where "p", "q", "in", "fi", "inc" are as in T1, and "s1" and "s2" are one of the separators ";" or "," then its expansion is given by Rule2

(Rule2)

```
|empty
|
|     if inc=0 or (fi>in and inc<0) or (in>fi and inc>0)
|
|substitute(p,i,in) [s1 p i q s2|in+inc,fi,inc] substitute(q,i,in)
|
|     otherwise
```

where "substitute(pattern,index,value)" is defined as in Rule1. An alternative shorter way of specifying the long conditional expressions for the empty expansion is

```
inc=0 or (fi-in)*inc<0
```

As we shall see in the next section the form T2 is not a valid form of replicators as it involves the "@" on both sides of the index placer " i ". Besides replicators of type T1, two other forms of replicators are valid, denoted by T2a and T2b which involve the "@" on one side of the index placer only:

T2a [p @ s[i]q |in,fi,inc] and
T2b [p[i]q @ s |in,fi,inc]

where "p", "q" are patterns as in T1, and "s" one of the separators ";" and ",". The expansion of replicators of the forms T1, T2a and T2b may be defined by one rule:

(Replicator Expansion Rule)

```
|if inc=0 or (fi-in)*inc<0 then empty
|
|otherwise
|
|for T1 :substitute(p,i,in) [p[i]q|in+inc,fi,inc] substitute(q,i,in)
|
|for T2a:substitute(p,i,in) [s p[i]q|in+inc,fi,inc] substitute(q,i,in)
|
|for T2b:substitute(p,i,in) [p[i]q s|in+inc,fi,inc] substitute(q,i,in)
```

The expansion of distributors was not formally defined directly; it was either described by an example or in terms of a replicator generating the same string. For example, in [LSB79] the expansion of the distributor

```
cs1(cs2(CN1 cs3 CN2(k3, ,) cs4 CN3(,k4,)))
```

where csi for i=1,...,4 are either ";" or "," and CNj for j=1,2,3 are collectivisors defined by

```
array CN1(n,m)
array CN2(k,n,m)
array CN3(n,k2,m)
```

was defined to be the same as the string obtained from

```
([[[(CN1(i,j)cs3 CN2(k3,i,j) cs4 CN3(i,k4,j))@cs2[i]|1,n,1])@cs1
                                                    [j]|1,m,1])
```

after all replicators are expanded.

After the expansion of all the replicators and distributors in a macro program and the elimination of collectivisors, the resulting

string should be a basic program.

After this informal presentation of the macro notation features, we next review various notations and subnotations in detail, focussing our attention on their formal grammars and indicating which parts of the notation may be extended to obtain a concise representation of more basic programs and which parts of the grammars should be modified to obtain a more precise formulation of what a macro COSY program may generate. It is recommended that the reader, and especially when not familiar with the macro COSY notation, should leave the section 3.1 until a later reading. In section 3.2 we propose a new notation and grammar for macro COSY which incorporates the suggestions for extensions and modifications of section 3.1. In section 3.3 we define and characterize the expansion of replicators and distributors and prove certain properties they possess. Some of these properties are used in proving that the expansion of any program produced by the grammar of section 3.2 may be produced from basic COSY rules as well. Finally in section 3.4 we evaluate the new notation and grammar.

3.1 A REVIEW OF MACRO COSY NOTATIONS

The macro notation has evolved considerably since it was first introduced [L76, LT77]. In this section we shall review the grammars for a number of notations and subnotations which have been used, concentrating our attention mainly on the syntax rules for collectivisors, replicators and distributors. In the syntax rules of this section we shall use the same meta-language conventions as in section 2.1.

3.1.1 The Macro COSY Program

A macro program consists of collectivisors, paths, processes, and replicators generating paths and processes, which are usually called bodyreplicators. According to the grammar in [L76, TL77] collectivisors, paths, processes and bodyreplicators appear between the word pair "begin" and "end". The syntax of a macro program is given by:

program=begin {{path/process/collectivisor/bodyreplicator} ?;}+ end

Later the word-symbols "begin" and "end" were replaced by "program" and "endprogram" respectively [L79, LSB79, LSC81] and the ";" was eliminated as a delimiter between paths, processes, bodyreplicators and collectivisors. Some grammars [LSC81] force the ordering that collectivisors should appear immediately after the word "program" followed by all paths and bodyreplicators generating paths which in turn are followed by all processes and bodyreplicators generating processes. This ordering restricts the ordering of paths and processes in basic programs obtained by expansion. The ordering of paths and processes in basic programs is not important. But the ordering specified in [LSC81] degrades the conciseness of macro programs in representing basic programs. Also the readability of macro programs is affected as the enforced ordering may not be the best way to group collectivisors, paths and processes, and bodyreplicators together. In [L79, LSB79] collectivisors were not used at all.

3.1.2 The Collectivisors

Collectivisors are used to declare subscripted operations of any finite number of dimensions. The collectivisor which declares subscripted operations corresponding to rectangular arrays the indices of which take consecutive positive integer values starting from 1 has been extensively used. Typical syntax rules may be found in [L76, TL77, TL78]:

collectivisor=array {collectivename @,}+({upperbound @,}+)
collectivename=upper-case-letter{upper-case-letter/digit/_}*
upperbound=integer-expression

The value of the integer expression "upperbound" should be greater than or equal to 1.

In [LS80, LSC81] the explicit specification of a lowerbound was permitted, thus increasing the class of subscripted operations which may be declared. The syntax of these replicators is given by:

collectivisor=array {arrayid @,}+({lowerbound:upperbound @,}+)

where "arrayid" is defined like "collectivename" above. The value of the integer expression "upperbound" should be greater than or equal to the value of the integer expression "lowerbound".

Subscripted operations which do not correspond to rectangular arrays and/or the indices of which are not consecutive integers could also be declared by the collectivisors. Replicators were used to specify either the exact set of admissible indices for each collective name or the exact set of admissible subscripted operations. The first approach was used in [LTD79]. For example, the subscripted operations

S(1,1),
S(3,1), S(3,2), S(3,3),
S(5,1), S(5,2), S(5,3), S(5,4), S(5,5)

would be declared by

C3 array S<[[(i,j)[j|1,i,1][i|1,5,2]>.

The second approach was used more extensively [LTD80, C80]. According to it, the subscripted operations S would be declared by

C4 array [[S(i,j)@,[j|1,i,1] @,[i|1,5,2].

Both approaches specify equally concisely a single collection of subscripted operations. The advantages of the second approach become apparent when two or more collections of subscripted operations are to be declared which have the same range in some of their dimensions. In [LTD79], for example two collections of operations GET and GR were declared by C5

C5 array GET<[[[(p,w,f)[p|1,m,1][f|w,n,1][w|1,n,1]>
array GR<[[(w,f)[f|w,n,1][w|1,n,1]>

in which GR and GET have the same index range in two dimensions but had

to be declared by two distinct collectivisors. In [LTD80] though, the same operations were declared by a single collectivisor C6

C6 array [[GR(w,f),[GET(p,w,f)@,[p]|1,m,1] @,[f]|w,n,1] @,[w]|1,n,1]

much more concisely.

Although collectivisors involving replicators were used extensively, no formal grammar was ever given for them.

3.1.3 The Bodyreplicators

As we have seen in example FB(n) in the introduction of chapter 3 specifying the n free frame buffer, replicators may generate collections of paths and/or processes. These replicators have been called "bodyreplicators" [L76, TL77, LTS79, LT78] or "replicatorprogrambody" [L79, LSB79] when they may generate paths and/or processes and "replpathprogrambody" and "replprocessprogrambody" [LSC81] when they may only generate paths and processes, respectively. We shall be refering to them as "bodyreplicators". The first syntax rule for them may be found in [L76, TL77]:

bodyreplicator=[{bodypattern @ separator [index]
/bodypattern1 [index] bodypattern2}|in,fi,inc]

bodypattern={{path/process}@;}+

where "in", "fi", "inc" are integer expressions and "index" is an identifier distinct from any operation in the program.

The "separator" in the first option of "bodyreplicator" should simply be ";" since the other separator, namely "," was never used at that position. Later the ";" was eliminated as a delimiter between paths and/or processes appearing only as the synchronization symbol for sequentialization of orelements in paths and processes. This option produces bodyreplicators generating consecutive regularities.

The second option of "bodyreplicator" produces bodyreplicators which generate imbrication of paths and processes. Since paths and processes simply follow each other and cannot be nested within each other, their imbrication was not essential and the same collection and ordering of paths and/or processes could be generated by bodyreplicators of the first option. For example the expansion of the bodyreplicator P13

P13 [path DEPOSIT1(i);REMOVE1(i)end|i|
path DEPOSIT2(i);REMOVE2(i)end |1,n,1]

could be generated by two bodyreplicators P14, P15

P14 [path DEPOSIT1(i);REMOVE1(i)end|i|1,n,1]
P15 [path DEPOSIT2(i);REMOVE2(i)end|i|n,1,-1]

or even by a single bodyreplicator P16

P16 [path DEPOSIT1(i);REMOVE1(i)end
path DEPOSIT2(n-i+1);REMOVE2(n-i+1)end|i|1,n,1]

To guarantee the well-formedness of the basic program obtained after the expansion of a bodyreplicator the meta-restriction MR1 was used:

MR1

"bodypattern1" and "bodypattern2" must be strings of symbols such that the omission of

[...|index|...|in,fi,inc]

yields a valid expression in basic COSY except for possible occurrences of indices.

Meta-restriction MR1 does not only exclude too wide bodyreplicators, but also some which generate well-formed basic strings. The reason is that paths and processes in "bodypattern1" and "bodypattern2" may involve replicators and distributors in their sequences which are not valid expressions in basic COSY. The meta-restriction MR1 was not really necessary when the second option in the rule for "bodyreplicator" is

replaced by:

bodypattern ; [index] bodypattern

which is precisely the syntax of the regularities in bodyreplicators generating imbrication.

The above rules do not permit nesting of bodyreplicators, but any number of paths and/or processes could constitute a bodypattern. The replicators in P4 and P5 are permitted under these rules (pgs 32,33, resp.)

In [LTS79] the ";" was eliminated as a delimiter between paths and/or processes. The syntax of all replicators was centred around one rule:

&replicator=[&pattern1 [index] &pattern2 |in,fi,inc]

where "&" is replaced throughout by one of "body" or " ". The non-terminal "bodypattern" was defined as follows:

bodypattern=body/bodyreplicator
body=path/process

To guarantee the well-formedness of the expanded program a meta-restriction was defined which when applied to bodyreplicators reduces to MR1. This meta-restriction is not necessary on bodyreplicators when the non-terminals "bodypattern1" and "bodypattern2" are defined as "bodypattern".

The above rules permit nesting of bodyreplicators. For example the bodyreplicator P17 is permitted:

P17 [[path TR(i,j);TR(i+1,j) end [i] |1,k+1,1] [j] |1,n,1]

specifying n pipelines of size k.

The grammars in [LSB79, L79] defined bodyreplicators, produced by the non-terminal "replicatorprogrambody" which is defined as follows:

replicatorprogrambody=programbody
/[replicatorprogrambody index |in,fi,inc]

programbody=pathprogrambody processprogrambody

pathprogrambody={path}*

processesprogrambody={process}*

According to the above rules bodyreplicators may be nested and any number of paths and/or processes could be in each one provided paths appear before processes. These replicators unlike the replicators in [LTS79] do not generate imbrication of paths or processes and no meta-restriction was necessary to be applied to them. However, the way "programbody" is defined permits the production of empty program bodies and empty regularities in bodyreplicators. Consequently, the expansion of macro programs may yield basic programs with empty bodies which are not permitted by the basic COSY syntax. This could be avoided if the rules for "programbody", "pathprogrambody" and "processprogrambody" are replaced by the rule:

programbody={path/process}+

The bodyreplicators produced by the rules in [L79, LSB79] above, always generate well-formed basic notation strings when their expansion is not empty.

In [LSC81] the syntax rules

replpathprogrambody={path/[replpathprogrambody index |in,fi,inc]}*

replprocessprogrambody={ process
/[replprocessprogrambody index |in,fi,inc]}*

produce bodyreplicators generating either paths or processes. The paths and bodyreplicators generating paths must appear before processes and bodyreplicators generating processes. Nesting of replicators generating paths and nesting of replicators generating processes is permitted but

nesting of one type inside the other is not. Similarly to the rules in [L79, LSB79] the above rules also permit empty program bodies and empty regularities in bodyreplicators.

3.1.4 The Paths and Processes

These differ from the paths and processes of basic COSY in that they may include replicators, distributors and indexed operations the indices of which may depend on replicator indices. Some of the grammars developed specify that they may appear as elements, others as orelements, and others as sequences. Here we examine the implications of each of these choices.

The first syntax for them appeared in [L76] where the following rules for paths and processes were given:

path=path pathsequence end

pathsequence=starsequence

starsequence=starsequence;starorelement/starorelement

starorelement=starorelement,starelement/starelement

starelement=pathelement*/pathelement

pathelement=element/(pathsequence)/pathreplicator

process=process sequence end

sequence=sequence;orelement/orelement

orelement=orelement,element/element

element=operation/(sequence)/replicator/distributor
/collectivename({{integer/integerexpression} @,}+)

In this grammar the "sequence" and "pathsequence" differ in that the former may produce starelements. However they both may produce replicators, produced from "replicator" if the whole string is produced by "sequence", or from "pathreplicator" if the whole of the string is produced from "pathsequence". The "pathreplicator" according to the above syntax could be starred. We believe that this makes the notation confusing, since after expansion the star only applies to the rightmost element of the resulting string and not to the whole string. However this choice does not generate invalid basic COSY programs and furthermore it increases the power for conciseness of the replicators. Consider for example path P18:

P18 path A(1);A(2);A(3)* end

which may be generated by P19

P19 path [A(i)@;[i]|1,3,1]* end

which is permitted by the syntax of [L76]. In these syntax rules the non-terminal "operation" produces only simple operations. Subscripted operations are produced by the last option of the non-terminal "element".

In the grammars of [TL77, LT78] "pathsequence" was replaced by "sequence" so both paths and processes may include replicators and distributors but no starelements.

The syntax rules in [LTS79] for paths and processes was given by

path=path sequence end

process=process sequence end

sequence=sequence;orelement/orelement

orelement=orelement,starelement/starelement

starelement=element/element*

element=operation/(sequence)/replicator/distributor
/collectivename({indexexpression @,}+)

According to the above rules starelements are reintroduced in the sequences of processes. Consequently, replicators and distributors may be starred as in the sequences of paths.

In [L79] various syntax rules were developed. Some of these rules however, specify both the context of replicators in sequences together with the syntax of the replicators themselves. We feel that these are two distinct issues. A replicator for example, could appear in a path as a sequence, as an orelement or as an element but in each case it may generate any string be it a sequence, or an orelement, or an element forming in each case a well-formed basic string. We took the liberty to split some of the rules so that we may concentrate on one of these issues at a time without being distracted by the other. By doing so we create some new non-terminals which we superfix by "0" to indicate that these were not in the original syntax of [L79] but are due to our modifications. In the next subsection we give the original syntax rules and examine what strings replicators generate and whether these strings on their own are legal basic COSY and only then we examine both the expanded string together with its context for well-formedness. Let us examine the syntax of the paths and processes in [L79]:

```
path=path {sequence/replicator^0} end
process=process {sequence/replicator^0} end
sequence={orelement @;}+
orelement={starelement @,}+
starelement=element/element*
element=operation/indexedoperation/(sequence)
```

As it was observed in [L79] this syntax only generates sequences in paths which consist of single replicators or a number of them individually nested within "( )". The above rules cannot produce replicators in other contexts as for example that of P20

```
P20 path a;[...];[...];b end
```

where "[...]" indicate replicators.

The second set of syntax rules given in [L79] replaced the production for non-terminals "path", "process" and "sequence" by

```
path=path sequence end
process=process sequence end
sequence={replicatororelement @;}+
replicatororelement={orelement/orreplicator^0}
```

respectively. According to the above rules a replicator can only appear as an orelement in a sequence and therefore only in the following contexts:

on its left any of	on its right any of
path	end
;	;
(	)

This implies that P20 is permitted but replicators cannot appear in the context of starelements as for example in path P21

P21 path a,[...],[...],b end

Certainly a sequence of the form in path P22

P22 path a,([...]),([...]),b end

is permitted but we should not conclude that this syntax just generates redundant parentheses maintaining the semantics of the path when expanded. It may well be that the additional parentheses change these semantics. It all depends on the main connective of the expanded string. If it is a comma then the parentheses are just redundant, but if it is a semicolon these may change the semantics of the path if one of the separators around the replicator is a ",". Let us consider the path P23

P23 path a,[C(i)@,$\boxed{i}$|1,3,1] end

which expands to P24

P24 path a,C(1),C(2),C(3),b end

the cycle set of which is

{a,C(1),C(2),C(3),b}

If the replicator in P23 were nested within parentheses as in P25

P25 path a,([C(i) @;$\boxed{i}$|1,3,1]),b end

the cycle set of its expansion would be exactly that of P24.

To demonstrate that additional parentheses may change the semantics of a path consider the path P26

P26 path a,[C(i) @;$\boxed{i}$|1,3,1],b end

which expands to the basic path P27

P27 path a,C(1);C(2);C(3),b end

the cycle set of which is

{a.C(2).C(3),a.C(2).b,C(1).C(2).C(3),C(1).C(2).b}

If the replicator in P26 were nested inside parentheses

P28 path a,([C(i)@;[i]|1,3,1]),b end

the cycle set of the path obtained by its expansion P29

P29 path a,(C(1);C(2);C(3)),b end

would be

{a,C(1).C(2).C(3),b}

which defines firing sequences different than those defined by the cycles of P27.

In the grammar of [LSC81] the syntax of paths and processes is given by:

path=path (gsequence)* end
processes=process (gsequence)* end
gsequence={gorelement @;}+
gorelement={gelement @,}+
gelement=element/replgseq/distrgseq
element=operation/indexedoperation/(gsequence)/element*

in which "gsequence", "gorelement", "gelement" stand for "generalized" sequence, orelement, element respectively as the strings they produce may include replicators and distributors. The non-terminals "replgseq" and "distrgseq" produce replicators and distributors respectively which appear in sequences. According to the above syntax, replicators and distributors appear as non-starred elements.

3.1.5 The Replicators in Sequences

In section 3.1.4 we examined the syntax rules which specify the context of replicators in sequences. Here we examine the syntax of these replicators.

The grammar in [L76] specified the following syntax for replicators in sequences:

```
&replicator=[{ &pattern @ separator [index]
             /&pattern1 [index] &pattern2}|in,fi,inc]
```

where "&" is replaced throughout by either "path" or " ". The non-terminals "pattern" and "pathpattern" were defined by:

```
pattern=sequence
pathpattern=pathsequence
```

The restriction MR1a, similar to MR1 for bodyreplicators, applied to patterns and pathpatterns:

MR1a

> "&pattern1" and "&pattern2" must be strings of symbols such that the omission of "[... [index] ...|in,fi,inc]" yields a valid expression corresponding to the prefix "&" of the patterns except for possible occurrences of indices.

The first option of the syntax rule for replicators produces replicators which generate consecutive regularities. The replicators produced by the non-terminals "pathreplicator" and "replicator" of this type always generate well-formed valid strings when expanded. This may be shown formally in the manner demonstrated in section 3.3 where we prove similar results for programs produced by the grammar of section 3.2. Furthermore they may generate regularities forming strings which may be produced by the basic COSY non-terminal "sequence". The disadvantage of this syntax is that the separators after the "@" are treated as of equal precedence, thus altering the precedence of comma over semicolon specified in basic COSY.

The second option intends to produce replicators which generate imbrication of regularities. Unfortunately, the replicators produced by this option, satisfying MR1a do not generate well-formed basic COSY strings. This is for two reasons:

1. Since replicators appear as elements and pathreplicators as pathelements in sequences, whatever they generate must be between one of the symbols "path", ";", ",", "(" on the left and ")", ",", ";", "end" on the right. Any legal string between these sets of symbols may in general, be produced by the non-terminal "sequence" of basic COSY.

2. The second reason is that the "@" does not appear in the second option at all, thus no separators are dropped upon expansion.

Let us consider the path P30

P30 path [(UP(i);RESET(i)[i])*|1,n,1] end

which is permitted by the syntax of [L76] and when expanded for n=3 generates the string

P31 path(UP(1);RESET(1)(UP(2);RESET(2)(UP(3);RESET(3))))end

which is not legal since there are some separators missing after RESET(1) and RESET(2). If we try to improve on that by putting a comma after RESET(i) in P30 obtaining P32

P32 path [(UP(i);RESET(i),[i])|1,n,1] end

and expand it again for n=3 we obtain the string

P33 path(UP(i);RESET(1),(UP(2);RESET(2),(UP(3);RESET(3),)))end

which would be legal if it were not for the comma after RESET(3). Only P30 is a legal pathreplicator according to the syntax and satisfies MR1a, but neither P30 nor P32 generate basic sequences. There is only one special case when P30 generates a well-formed expansion that is when

n=1 i.e. when the replicator generates one copy only

P34 path (UP(1);RESET(1)) end.

As we have noted replicators according to the syntax of [L76] appear as elements in a sequence. For this reason another meta-restriction should be imposed on them to exclude replicators specifying empty index ranges which would imply empty expansions and collision of terminal symbols in the context of the replicators. This should apply to all replicators which appear as sequences, orelements and elements. For example, if in the path P35

P35 path a,[C(i)@,|i|1,n,1] end

the value of n were zero, the path after the expansion of the replicator would be P36

P36 path a, end

which is not legal in basic COSY, as there is a collision of the terminal symbols "," and "end".

In [TL77] the "pathsequence" was replaced by "sequence". The syntax rules were simplified after the elimination of "&" standing for either "path" or " " but still all the previous comments regarding the grammar of [L76] apply to the grammar of [TL77] as well.

In [LS77] no formal grammar was given. A replicator was defined to be

> "an iterative copy operator which permits the finite representation of program text of finite but indefinite length".

The general form of a replicator was defined as

either [pattern |index| |in,fi,inc]
or [pattern1 |index| pattern2|in,fi,inc]

where the patterni for i=1,2 were defined to be strings. This is the most general replicator which may be defined. Obviously all possible regularities could be generated by using such replicators not necessarily forming well-formed basic strings. For this reason the meta-restriction MR2 was used:

MR2

patterni's must be such that the resulting string after expansion must be a valid expression in basic COSY.

The above rule may be interpreted in two ways. The first may be that the expansion of each replicator must be a sequence, or an orelement, or an element. However, the path in page 16 of [LS77]

```
P37 path
      test0*
     ,( countincr
      ;test1*[,(countincr;test*[i];countdecr)*|n,1,-1]
      ;countdecr
     )*
    end
```

contradicts this interpretation since the replicator itself does not expand to any valid expression in basic COSY.

The above path is consistent with the second interpretation by which programs may involve replicators in any context and patterni for i=1,2 may be any strings. For the replicator to be well-formed though, the string obtained after the expansion of the replicators must be well-formed basic COSY programs. This interpretation of MR2 has some interesting implications. In the introduction of this chapter we presented the replicator P4 specifying the three free frame buffer from which a generalization for an n-free frame buffer was derived by just altering the upper limit of the values the index takes from 3 to n. The second interpretation of rule MR2 does not in general permit this kind of generalization. Consider for example, the path P38

```
P38 path (([A(i);B(i))@,[i]|1,2,1] end
```

which is well-formed according to the second interpretation of MR2, since after the expansion of the replicator path P39

P39 path((A(1);B(1)),(A(2);B(2))end

is obtained, which is well-formed in basic COSY. If we generalize the replicator in P38 to generate n regularities of "A(i);B(i))," by replacing 2 by n path P40 is obtained:

P40 path (([A(i);B(i))@,[i]|1,n,1] end

The resulting string after the expansion of the replicator in P40 will only be well-formed when n=2. When n<2 there will be more opening parentheses than closing ones and when n>2 more closing parentheses than opening ones. Therefore the fact that a replicator is expanded to a well-formed basic string for a particular index range does not necessarily imply that this replicator will generate well-formed basic strings for any index range. We shall call this kind of replicator range dependent. We feel that these replicators should be avoided and that the macro notation should only allow replicators which when expanded always generate well-formed basic COSY strings for any non empty range of their index.

The syntax rules in the grammar of [LTS79] producing replicators which generate imbrication of regularities were similar to those presented in [L76] but the problem of not generating well-formed basic COSY strings is overcome. The syntax for the replicators generating consecutive regularities was considered as a special case of the replicator generating imbrication of regularities. The rules for the syntax of replicators were:

replicator=[pattern1 [index] pattern2|in,fi,inc]
pattern={sequence/separator}* /pattern @{;/,}
separator=;/,/*/(/)

where exactly one of patterni for i=1,2 must have the form "pattern @{;/,}". Similarly to [L76, LT79] a meta-restriction of the type of MR1 was applied to these patterns:

MR1'

"patterni" must be strings of symbols such that the omission of

[index |in,fi,inc] or
[index @{;/,}|in,fi,inc] or
[@{;/,} index |in,fi,inc]

yields a valid expression.

This syntax together with the meta-restriction MR1' produce replicators which generate well-formed basic sequences. This may be proved in the style we proved similar results in section 3.3. This syntax does not produce any range dependent replicators. The meta-restriction rule MR1' excludes all replicators which when expanded do not generate well-formed basic COSY strings. Furthermore the legal replicators may generate nested regularities. For example the paths

P41 path [(DEPOSIT(i)@; i ;REMOVE(i))*|1,n,1] end
P42 path [(DEPOSIT(i); i REMOVE(i))*@;|1,n,1] end

are both valid. However there is a class of nested regularities which they cannot generate. Consider for example the basic path P43

P43 path(A(1),(A(2),(A(3);B(3)),B(2)),B(1)) end

the sequence of which cannot be generated by any of these replicators. If we examine P43 we see that the element "(A(3);B(3))" is nested inside the element "(A(2),...,A(2))" which in turn is nested inside the element "(A(1),...,B(1))". The reason this kind of replicator cannot be written is not because the innermost element is not an exact copy of the other elements. This is true in general for any nested regularities. Consider for example the path P44

P44 path(A(1),(A(2),(A(3);B(3));B(2));B(1))end

in which the innermost regularity is "(A(3);B(3))" whilst the others are of the type "(A( ),...;B( ))". However the sequence of path P44 may be

generated by P45

P45 path [(A(i)@,[i];B(i))|1,3,1] end

We may characterize the class of regularities which cannot be generated by replicators. It is the class of regularities in which all are of the form

p(i) s1 ... s1 q(i)

in which "p", "q" are patterns involving some index "i" and "s1" represents one of the separators "," or ";", except for the innermost regularity which is of the form

p(fi) s2 q(fi)

in which the index "i" in the patterns "p", "q" is replaced by i's last value, namely "fi" and the separator "s2" is distinct from "s1". For a replicator to generate this kind of regularities, the separator after the "@" should not be stripped but should be replaced by another separator.

We have pointed out that the syntax rules of [LTS79] together with MR1' produce replicators which only generate well-formed basic COSY strings. Without MR1' though their syntax would be too wide as the strings the replicators could generate would not in general be well-formed in basic COSY. We feel that close-fitting formal syntax rules should be derived producing replicators which after expansion generate well-formed basic COSY strings. The syntax in [L79] and [LSC81] gave some partial solutions to this problem.

In [L79] the problem of more "close-fitting" rules for replicators was discussed and various syntax rules were developed. The approach followed was to start from syntax rules producing simple replicators and to extend them to produce replicators able to generate larger classes of regularities. The first replicator together with its context was defined by

path=path {sequence/[element @ separator i |in,fi,inc]} end
process=process {sequence/[element @ separator i |in,fi,inc]} end
separator=;/,

According to this definition nesting of replicators is not permitted. Replicators may generate sequences if the separator after the "@" is ";" or orelements if this separator is a ",". Since the regularity they replicate is "element", redundant parentheses have to be introduced when orelements or sequences are replicated. Consider for example paths P46 and P47

P46 path A(1),B(1);A(2),B(2);A(3),B(3) end
P47 path A(1);B(2);A(2);B(2);A(3);B(3) end

According to the above rules the paths P48 and P49

P48 path [(A(i),B(i))@; i |1,3,1] end
P49 path [(A(i);B(i))@; i |1,3,1] end

are permitted, involving replicators which when expanded generate paths with the same semantics as P46 and P47 respectively, by introducing redundant parentheses. The above rules cannot produce replicators which expand to precisely the paths P46 and P47.

The above syntax rules produce replicators which always generate well-formed basic COSY strings when their expansion is not empty. However, they may only appear in a very limited context, namely in place of whole sequences between "path" and "end" or between "(" and ")". Thus, as it was pointed out in the example (E20) in [L79] the process P50

P50 process b;[(AB(i);AE(i))@, i |1,n,1];c end

is not permitted. The syntax rules were then extended to cover at least this case:

```
path=path sequence end

process=process sequence end

sequence={ {orelement
          /[orelement @ separator[i]|in,fi,inc]}_@;}+
```

These rules also do not permit nesting of replicators. They do not introduce as many redundant parentheses as the previous rules though. For example P46 may be exactly generated by P51:

```
P51 path [A(i),B(i)@;[i]|1,3,1] end
```

The main limitation with both the above sets of rules in [L79] is that they do not permit nested replicators. As the example (E22) in [L79] demonstrates, the process P52

```
P52 process b;[[(AB(i,j);AE(i,j))@,[i]|1,n,1]@,[j]|1,k,1];c end
```

is not permitted. The syntax rules were extended to permit nesting of replicators:

```
sequence={replicatororelement @;}+

replicatororelement=orelement
                   /[replicatororelement @ separator[i]|in,fi,inc]

separator=;/,
```

The replicators produced by these rules generate well-formed basic COSY strings. Again, redundant parentheses have to be introduced when sequences are replicated as in P49. The intention in [L79] was not just to define replicators which generate well-formed strings. In addition the syntax in [L79] was aiming to define replicators the expansion of which could be produced by the basic COSY non-terminal "sequence" or "orelement" and this expansion to appear in a sequence as a subsequence or as a suborelement respectively. In other words the first and the last element of the expansion should bind with the rest of the expansion

and not with other elements in the rest of the sequence. Consider for example the path P53:

P53 path b;[A(i)@,$\boxed{i}$|1,3,1];c end

which when expanded yields P54

P54 path a;A(1),A(2),A(3);c end

The expansion of the replicator in P53 on its own yields an orelement and it is also an orelement in the context it appears in path P54. We will say that the replicator in P53 generates a syntactically strong string. Not all replicators generate syntactically strong strings and furthermore not in any context. Consider for example P55:

P55 path b,[A(i)@;$\boxed{i}$|1,3,1],c end

in which the expansion of the replicator on its own is a sequence. But its expansion in the context of P55

P56 path b,A(1);A(2);A(3),c end

is not a syntactically strong string since A(1) binds with operation b and A(3) with c and not with the rest of its expansion. The aim of [L79] was therefore to obtain syntax rules for replicators which generate syntactically strong strings in the context they appear. The previous syntax rules of [L79] do not permit path P55. They however permit path P57

P57 path [[A(i,j)@;$\boxed{i}$|1,2,1]@,$\boxed{j}$|1,2,1]end

in which the inside replicator does not produce syntactically strong strings, as may be seen when both replicators are expanded:

P58 path A(1,1);A(2,1),A(2,1);A(2,2) end

For this reason the definition of "replicatororelement" was redefined as:

```
replicatororelement=orelement
                    /([replicatororelement @;[i]|in,fi,inc])
                    /[replicatororelement @,[i]|in,fi,inc]
```

and at the cost of redundant parentheses it was simplified to:

```
replicatororelement=orelement
                    /([replicatororelement @ separator[i]|in,fi,inc])
```

These rules produce replicators which when expanded produce well-formed and syntactically strong strings. Their only disadvantage is that they introduce reduntant parentheses in three contexts. The first is when they appear as orelements in a sequence as for example in P59:

```
P59 path ([...]);([...]);...end
```

The parentheses are redundant for whatever strings the replicators generate. The second context is when replicators appear as elements in an orelement and the replicators generate orelements as in P60:

```
P60 path ([...@,[]|...]),([...@,[]|...]),...end
```

Finally, they introduce reduntant parentheses around the regularities they generate in the context

```
P61 path ...[(...;...;...)@;[]|...]... end
```

In the next chapter, where we address the problem of finding the semantics of a basic program generated from a macro program, directly from the macro program itself, we derive syntax rules for replicators generating syntactically strong strings without the enforcement of redundant parentheses.

The rules given in [L79] which we examined up to now, produce replicators which generate consecutive regularities. The first rule for replicators generating imbrication of regularities is:

```
replicatororelement=orelement
                    /([replicatororelement @ separator i
                       {/separator replicatororelement}|in,fi,inc])

                    / ([replicatororelement separator i
                          replicatororelement @ separator|in,fi,inc])
```

This rule produces all replicators produced by previous rules in [L79]. It also produces replicators which generate some imbrication of regularities as for example the replicator in P62:

```
P62 path ([A(i)@;i;B(i)|1,3,1]) end
```

which expands to

```
P63 path(A(1);A(2);A(3);B(3);B(2);B(1)) end
```

This is a special kind of imbrication. The general kind of imbrication of regularities is when these have opening parentheses on the left of the place holder "i" and corresponding closing parentheses on the right of the place holder as in the stack example P11, P12. To produce this kind of replicator the rules for "replicatororelement" were extended to

```
replicatororelement=
    orelement

   /([{rseparator}* replicatororelement @ separator i
     {/{rseparator}* replicatororelement {rseparator}*}|in,fi,inc])

   /([{rseparator}* replicatororelement @ separator i
    {rseparator}* replicatororelement @ separator|in,fi,inc])


rseparator=separator/(/)/*
```

Although these rules permit the replicators in P11, P12 they may also produce other replicators which do not generate well-formed basic COSY strings. For example, the number of opening parentheses on the left

hand side of the place holder "□" do not necessarily match with closing parentheses on the right hand side of the place holder. Therefore together with the above rule a meta-restriction rule is needed to filter out all those replicators which generate invalid basic COSY strings. It is apparent that even more close-fitting rules are needed.

In [LS80] the grammar for replicators was given by:

basicsymbol =some finite set of basic symbols not including the "@".

index =some possibly infinite set of symbols distinct from basic symbols

indexexpression =integer expression involving only indices and integer constants

pattern ={basicsymbol/index}*/replicator

replicator =[pattern{@{;/,}/}|i|pattern{@{;/,}/}| indexexpression,indexexpression,indexexpression]

Since no other rule for constraining the patterns of the replicators was given these may generate any strings of basic symbols. We feel that these rules are very wide and more close fitting rules are required improving on the syntax of [L79].

In [LSC81] we presented context-free syntax rules general enough to produce all the macro programs in this paper involving replicators generating well-formed basic COSY strings. The context-free rules were:

```
replgseq=[{gseqrepl1/gseqrepl2}|in,fi,inc]

gseqrepl1=gsequence @ sep [index]

gseqrepl2=gsequence @ sep [index] sep gsequence
        /{gsequence sep/} elementrepl {/sep gsequence}

elementrepl=elementrepl*
          /({gsequence @ sep [index] / [index] gsequence/gseqrepl2})
```

If we eliminate the middle option of the second alternative for "elementrepl" all replicators produced generate well-formed basic COSY strings. This again may be proved in the style we proved similar results in section 3.3. The above rules permit replicators which generate a large class of imbrication of regularities such as the stack specified by P11 and path P64:

```
P64 path [(A(i)@ ;[i]),B(i)|1,3,1] end
```

The above rules specify that any number of unmatched opening parentheses on the left of the place holder, match with closing parentheses on the right of the place holder. These rules however, cannot produce replicators which involve the "@" on the right of the place holder like the replicator in path P65

```
P65 path [SK(i),([i]A(i);B(i))@;|1,n,1] end
```

The rules of [LSC81] may be extended to permit such replicators:

```
replgseq=[gseqrepl|in,fi,inc]

gseqrepl=gsequence @ sep [index]
        / [index] gsequence @ sep
        /gsequence @ sep [index] sep gsequence
        /gsequence sep [index] gsequence @ sep
        /elementrepl

elementrepl=elementrepl*/(gseqrepl)
```

Although the above rules are close-fitting and all replicators they produce generate well-formed basic COSY strings they specify a mixed precedence of ";" and ",". The above rules may not produce replicators generating strings such as the sequence of path P43 since the "@" only strips and does not replace any separators by others. The need is still apparent for context-free rules producing more general replicators which expand to well-formed basic COSY strings.

3.1.6 The Distributors

Historically, distributors were the first macro feature to be used [CL76] in the path notation [LC75] as a shorthand. The term "distributor" was introduced later [L76] with the rest of the macro notation. In [LC75] the formal definition of the path definition was introduced and was extended by a SIMULA class-like construct which permits classes to contain both paths and processes. In [LC76] arrays of classes could be declared which were called sets. The shorthands

P.(,S) and P.(;S)

were defined, where S is a set containing k elements and P is an operation, called a procedurename in [LC76], in paths or processes in each class S(i) for i=1,2,...k. These shorthands denoted the strings:

P.S(1),P.S(2),...,P(k) and
P.S(1);P.S(2);...;P.S(k) respectively.

In [L76, TL77] the notation for distributors was changed to deal with collective names and not with sets:

```
distributor=
        separator({{collectivename
              /collectivename({{integer/indexexpression}@,}+)}
                                    @separator}+)
```

A distributor may only generate certain types of consecutive

regularities. The indices needed in each of these regularities to generate subscripted operations are implicitly generated. These are the same as the indices in a dimension of the collective names involved in a distributor, as defined by the collectivisor. When a distributor involves collective names with more than one dimension then it is required to specify the dimensions over which the connectives are to be distributed by leaving a blank field in their index list. If a collective name has all its index fields blank then its index list may be eliminated altogether. This applies to all distributors and will be assumed to apply throughout this section.

In [L76, TL77] the following constraint was imposed on distributors:

> the sets of indices corresponding to each of the dimensions over which collective names are to be distributed must be the same, otherwise the distributor is not well-formed.

If this constraint is satisfied then we say that the dimensions to be distributed are compatible. Sometimes the above constraint is refered to as the compatibility criterion. According to the compatibility criterion the distributors D1, D2 and D3

```
D1 ;(DEPOSIT)
D2 ,(DEPOSIT;REMOVE)
D3 ,(A(2, ))
```

are well-formed, provided the collectivisors C7 and C8

```
C7 array DEPOSIT,REMOVE(n)
C8 array A(2,2)
```

have been declared. When a distributor is expanded each regularity is wrapped between an opening and a closing parentheses. The distributor D1 then expands to

```
(DEPOSIT(1));(DEPOSIT(2));...;(DEPOSIT(n))
```

the distributor D2 to

```
 (DEPOSIT(1);REMOVE(1))
,(DEPOSIT(2);REMOVE(2))
          ...
,(DEPOSIT(n);REMOVE(n))
```

and D3 to

```
(A(2,1)),(A(2,2))
```

The regularities that distributors generate may be produced by the non-terminal of basic COSY "sequence", but do not however include elements of the type "(sequence)". As may be seen in the strings generated by D1 and D3, the parentheses enforced around each regularity may be redundant.

Later in [LTS79] nested distributors were defined such as

```
D4 ,(;(A))
```

where A is assumed to have been declared by the collectivisor C8.

For this distributor we must specify which separator applies to which dimension of the collective name. The adopted convention was that the innermost separator will apply to the leftmost dimension, the next separator to the leftmost not allocated dimension etc. Obviously, a collectivisor must have as many dimensions to distribute over as the number of nested distributors it is in. The distributor D4 therefore expands to

```
((A(1,1));(A(2,1))),((A(1,2));A(2,2)))
```

provided A has been declared by the collectivisor C8. The syntax of the distributor was defined in [LTS79] by:

```
distributor={;/,} { distributor
                  /({{collectivename
                   /collectivename({{integer/indexexpression}@,}+)}
                       @separator}+)
```

The syntax of [LSC81] permits the production of distributors which generate any consecutive regularities the elements of which may be starred and/or which could be of type "(sequence)". The syntax of distributors was given by:

distributor={;/,}[gsequence]

In [LSC81] the lower bound of the collective names were explicitly specified and not implicitly fixed to 1. The compatibility criterion of [L76] was accordingly relaxed by requiring

> the sets of indices of dimensions of collective names to be distributed by the same collectivisor, to have the same number of elements.

For example if the collective names A, B, C and D were defined by:

```
C5 array A,B(2)
   array C,D(2:3)
```

the distributor

```
D6 ;((A;B*;C),D)
```

would expand to

```
 (A(1);B(1)*;C(2)),D(2)
;(A(2);B(2)*;C(3)),D(3)
```

In this syntax replicators were permitted inside distributors as well as distributors inside replicators, as in the earlier grammars. In that sense distributors and replicators become symmetrical.

The expansion of distributors, unlike the expansion of replicators, was never formally defined directly. Their expansion was either described by an example or in terms of a replicator generating the same string. Furthermore, the expansion of distributors was not at all defined when distributors involve collective names corresponding to

non-rectangular arrays.

3.1.7 Some More Replicators

In the review up to now we have examined various replicators generating subscripted operations, paths and/or processes, sequences, orelements, starelements and elements. However, the values of their indices always formed finite arithmetic progressions.

In macro COSY other replicators have been defined which permit the index to take values forming infinite arithmetic progressions and others which take a finite number of values but do not form arithmetic progressions in general.

In [SL77, SL79] it was shown that any "program" using the extended semaphore primitives (ESP's) of Agerwala [A77] as its only means of synchronization and which is in some sense "bounded" has an equivalent description in the COSY formalism. It was pointed out however that to obtain a complete translation of a given ESP program, which may contain unbounded semaphores, requires a real extention of the descriptive power of the COSY notation as it may only describe finite systems. Such an extention was suggested in terms of Petri-nets [P76] in [SL77] and in terms of the "Cyc" operator in [SL79] but as it was pointed out, out of theoretical interest. After this extension infinite counters were defined

P66 path [(V(s)@;$\boxed{i}$;P(s))*|1,∞,1] end

which were given vector firing sequence semantics.

In [LTD79, D79, LSB79, LTD80, SL80] replicators were defined the index of which could take a finite number of values not necessarily forming arithmetic progressions. In [LTD80] these were called test replicators. Two formalisms were used, both incorporating predicates to select or define the range of the index. The replicators in [LTD79, D79, LTD80] used predicates to select the range of an index out of an arithmetic progression. For example path P67

P67 path [SIEVE(i)@; i, i is prime |2,15,1] end

generates P68

P68 path SIEVE(2);SIEVE(3);SIEVE(5);SIEVE(7);SIEVE(11);SIEVE(13) end

filtering out the elements in the arithmetic progression 2,3,...,15 which are not prime. The standard replicators therefore may be viewed as a special case of the test replicators in which the value of the predicate P(i) is true for all values of the replicator index "i" takes.

In [SL80] the predicates were defined outside the replicators. These predicates were used in place of the arithmetic progression generator "|in,fi,inc" generating the integers which satisfied them. For example the replicator in P50 would have been written as

predicate P(i)=(2≤i≤15 and i is prime)
P69 path [SIEVE(i)@ ; i|P(i)] end

This concludes the review of most macro COSY notations and subnotations. In the next section we introduce a new macro notation which improves or eliminates altogether drawbacks of the notations we examined in this section.

3.2 A NEW NOTATION AND GRAMMAR FOR MACRO COSY

In this section we make some changes to the macro COSY notation improving the readability of macro programs. We extend it in such a way that new replicators are included, generating classes of basic COSY strings which cannot be generated by replicators produced by the grammars reviewed in section 3.1, and in such a way that new

distributors are included, generating more basic COSY strings more concisely than replicators. Together with the notation we present the syntax rules for producing macro programs in this new macro COSY notation. Our general considerations in developing the new notation and grammar were mainly four:

1. The syntactic well-formedness of a macro COSY program produced by the grammar should imply the syntactic well-formedness of the corresponding basic COSY program resulting from expansion. Our aim is to derive formal context-free rules avoiding meta-restriction rules on the regularities of replicators. For this reason we need to specify exactly what we are allowed to write in replicators and distributors and where these should appear in the programs.

2. The grammar should be general, producing replicators and distributors able to represent a large class of regularities of structures concisely.

3. The grammatical rules should be uniform with the rules for basic COSY and should formally show the hierarchy of the macro COSY notation over the basic COSY notation.

4. The macro elements should represent the regularities they generate in a way as obvious as possible for the reading of macro programs to be possible without their formal expansion.

The meta-language conventions which will be used in the syntax rules in this section will be the same as in last section. The subsections are divided in the same way as in the last section; in each we examine a major syntactic category.

3.2.1 The Macro Program

A macro COSY program will consist of collectivisors, paths, processes and bodyreplicators appearing between the word pair "program" and "endprogram". Since after expansion of a macro program all its

collectivisors disappear, the macro program should include at least one of a path, process, or bodyreplicator, for the body of the basic program obtained by expansion to be non-empty. The syntax for macro programs is given by:

MN1. mprogram=program mprogrambody endprogram
MN2. mprogrambody={{collectivisor}* {mpath/mprocess/bodyreplicator}}+

In the above rules, and henceforth, non-terminals of macro COSY which correspond to non-terminals of basic COSY have been obtained by prefixing the latter by "m" standing for "macro".

According to the above rules, collectivisors, macro paths, macro processes and bodyreplicators may appear in any order, with the exception that no collectivisor may appear after all the paths, processes and bodyreplicators. The following restriction is imposed on programs:

(MPrest)

Collective names should be declared before any path or process involving any of its subscripted operations.

The context-sensitive restriction (MPrest) is imposed so that for the expansion of a macro program one pass is sufficient. It also makes the syntax checking more efficient as well. For otherwise, two passes would be required for expansion and syntax checking, since the collective names and the number of their dimensions have to be known in either case and in addition, when expanding, the bounds of the indices in every dimension have to be known as well. We could avoid this meta-restriction by forcing all collectivisors to appear before paths, processes and bodyreplicators. We would however need the context-sensitive restriction that all subscripted operations in macro paths and macro processes should be permitted by the collectivisors. When writing programs though, we find it sometimes convenient to declare collective names near the paths which involve indexed operations corresponding to these collective names.

3.2.2 The Collectivisors

Previous notations permit declarations of collective names corresponding to rectangular arrays and to arrays of other shapes. In the new notation we shall permit both types of collective names to be declared. When declaring rectangular arrays two conventions have been followed: either the lower bound of indices in their dimensions are implicitly considered as having the value one, or the lower bound is explicitly specified. In the new macro notation we combine both conventions. We also follow the convention that collective names will be in upper case letters. When subscripts in the dimensions of arrays are consecutive positive integers starting from 1, the lower bound in these dimensions may be implicitly assumed to have the value one and only the upper bound has to be specified. We permit collective names with a different number of dimensions and/or different bounds in their dimensions to be declared by the same collectivisor. Two notational changes are introduced in declaring collective names:

1. The elimination of commas between collective names. The intention is to confine the use of the comma to sequences, as the synchronization symbol for "choice", as much as possible.

2. The introduction of the word symbol "endarray" which indicates the end of a declaration. All declarations will now be enclosed between word symbol pairs "array" and "endarray" in the same way major syntactic entities like "programbody" and "sequence" in basic COSY are enclosed between word symbol pairs.

For example the declaration NC1

```
NC1 array A(k) endarray
    array B C(5) D(3,m) endarray
```

declares the subscripted operations

A(1),...,A(k),
B(1),...,B(5),
C(1),...,C(5),
D(1,1),...,D(1,m),
D(2,1),...,D(2,m),
D(3,1),...,D(3,m)

The "N" in front of the mnemonic names of examples in this section and in section 3.3 indicate that these are written in the new macro notation introduced in this section. The letter "C" indicates collectivisors, the letter "P" paths, processes or bodyreplicators and the letters "D" and "R" distributors and replicators in sequences, respectively.

If the lower bound in some dimensions of collective names is not 1 but some other fixed integer n we may specify it explicitly as in [LSB79, LSC81]. To specify for example that the single dimension of collective name E has lower bound n and upperbound k, and that the first dimension of the two dimensional collective name F has lower bound m and upper bound n and its second dimension lower bound one and upper bound k, we write:

NC2 array E(n:k) F(m:n,k) endarray

We may also combine the declarations in NC1 and NC2 in one declaration:

NC3 array A(k) B C(5) D(m,3) E(n:k) F(m:n,k) endarray

For the collectivisors to be well-formed we shall require all the declarations to satisfy the collectivisor restriction Crest1:

(Crest1)
the upperbound of the dimensions of the collective names has to be greater than or equal to their corresponding implicit or explicit lowerbound.

We permit also declaration of subscripted operations the indices of which either are not consecutive integers or depend on the index of some

other dimension. For example, the index in the first dimension of the subscripted operations:

S(1,1)
S(3,1), S(3,2), S(3,3)
S(5,1), S(5,2), S(5,3), S(5,4), S(5,5),

takes values 1, 3, 5 and the index in their second dimension takes consecutive values from one to the value of their first dimension. We shall use replicators to generate the set of admissible subscripted operations as in [LTD79]. The subscripted operations corresponding to S may be declared by the collectivisor NC4

NC4 array #i:1,5,2[#j:1,i,1[S(i,j)]] endarray

using replicators the notation of which we have modified. We have changed the generator for index values "$\boxed{i}$|in,fi,inc" to "#i:in,fi,inc" and moved it in front of "[ ]". The reason for the change was more or less technical: the place holder "$\boxed{\ }$" is not standard in size depending on the length of the index identifier and it is not a standard character symbol in any computer or typewriter. It has always to be drawn by hand on paper and be replaced by other symbols whenever a macro program is to be given as input to a computer program, for example for syntax checking or for expansion. The reason we moved the index generator in front of "[ ]" is mainly for the improvement of readability of replicators. The replicator may now be read from left to right as

> "index i takes values from in to fi in steps of inc which upon expansion are replacing index 'i' in each copy of the regularity inside '[ ]' ".

Thus we have separated the index specification part which is common to all replicators no matter what they generate, from the regularities they generate, which are now the only strings in "[ ]". Similar notational changes will be applied to bodyreplicators and replicators appearing in sequences. For the replicators to be well-formed they should obey the second collectivisor restriction Crest2:

(Crest2)

Each replicator must specify a non empty range for its index.

Restriction Crest2 guarantees that replicators in collectivisors declare at least one subscripted operation corresponding to each collective name.

Subscripted operations with the same subscripts in all their dimensions may be declared by the same replicators and will not be separated by commas, simplifying the syntax of these replicators and eliminating the comma between subscripted operations altogether. Also subscripted operations with the same subscripts in some of their dimensions, may be groupped together in the same replicator, e.g.

NC5 array #i:1,5,2[T(i) #j:1,i,1[S(i,j) U(i,j)]] endarray

where the collective name T corresponds to the operations

T(1), T(3), T(5)

and the collective name U to the operations

U(1,1),
U(3,1), U(3,2), U(3,3),
U(5,1), U(5,2), U(5,3), U(5,4), U(5,5)

The subscripted operations in replicators may be indexed by expressions involving replicator indices. These expressions should satisfy the third collectivisor restriction Crest3:

(Crest3)

All expressions indexing collective names should yield integers for all the values which the indices they involve take.

We also permit grouping together subscripted operations, indexed by expressions depending on replicator indices indexing other subscripted

operations, as in NC6:

NC6 array #i:1,5,2[V(i+3)#j:1,i,1[S(i,j)]] endarray

where V corresponds to the operations V(4), V(6), V(8). We shall require that collectivisors involving nested replicators, are constrained by the fourth collectivisor restriction Crest4:

(Crest4)

A collectivisor involving nested replicators must be of the form

#kn:inn,fin,incn[...#k1:in1,fi1,inc1[Y(h1,h2,...,hn)]...]

where hi for i=1,...,n are expressions involving indices kj for j=1,...,n such that each ki for i=1,...,n must appear in at least one dimension, and an index ki i=1,...,n may only appear together with indices kj for j>i in a single expression and in at most (i-1) expressions with indices kj for j<i.

The restriction Crest4 is imposed to guarantee the independence of the indices of different dimensions of the same collective name and to avoid duplication of declarations of subscripted operations. The invalid declaration

array #i:1,5,2[W(i,i+1)] endarray

declares a two dimensional array W corresponding to the subscripted operations:

W(1,2), W(3,4), W(5,6)

The indices in the dimensions of W are dependent for if one index is known the other may be determined. This type of declaration contradicts the notion of dimension and for this reason is excluded. Crest4 also excludes duplication of declaration of subscripted operations as for example the following invalid collectivisor specifies:

array #i:1,5,2[#j:1,i,1[T(i) S(i,j)]] endarray

which declares T(3) three times and T(5) five times. There is a third type of collectivisor which is excluded by Crest4 which does not define

dependent dimensions nor dublicates subscripted operations such as

array #i:0,9,1[#j:0,9,1[A(100*j+i)]] endarray

The reason we have excluded this type of collectivisor is more subtle and has to do with the expansion of distributors. We shall discuss this point in section 3.4 having examined the distributors and their expansion. We may characterize the shapes of arrays declared by replicators as being finite n-dimensional arrays, the indices in each dimension of which are generated by an integer expression depending on integer variables taking values from an arithmetic progression. (*)

We may also combine the NC3 and NC6 types of declarations in a single declaration. The complete syntax for the collectivisors is:

MN3. collectivisor=array {simpleardecl/replardecl}+ endarray
MN4. simpleardecl={arrayid }+({{iexpr:/} iexpr @,}+)
MN5. replardecl=index_spec[{replardecl/arrayid({iexpr @,}+)}+]
MN6. index_spec=#index:iexpr,iexpr,iexpr
MN7. arrayid=uc-letter{uc-letter/digit/_}*

where "simpleardecl" stands for a list of collective names which correspond to simple rectangular arrays together with their bounds and "replardecl" stands for the replicator generating admissible sets of subscripted operations. The non-terminal "index_spec" stands for the index specification part of a replicator and "iexpr" for an integer expression. The syntax of "index" is the same as that of a simple operation rule BN9 of basic COSY. Identifiers used for replicator indices though, must satisfy the index restriction Irest1:

(Irest1)
Identifiers for replicator indices should be distinct from any identifiers used for simple operations.

and the restriction Irest2:

(*) The following restriction must also hold:

(Crest5)
An array identifier may only occur once in collectivisors.

(Irest2)

Replicator indices are only defined inside "[ ]" of the replicator with which they are associated. In the scope of a replicator index no other replicator index having the same identifier is permitted.

The restrictions on replicator indices (Irest1) and (Irest2) apply to all replicators.

3.2.3 The Bodyreplicators

We permit replicators, bodyreplicators as we call them, which may generate paths and/or processes. Bodyreplicators are permitted to generate consecutive regularities of paths and/or processes. We also permit nesting of bodyreplicators. The only change to the grammar of [LTS79] is a notational one and involves the index specification part of the bodyreplicator, which was changed from " index |in,fi,inc" to "#index:in,fi,inc" and was moved in front of "[ ]". Their syntax is formally given by:

MN8. bodyreplicator=index_spec[{mpath/mprocess/bodyreplicator}+]

No meta-restriction is needed to guarantee the well-formedness of the expanded programs. If each of the paths and processes they generate is well-formed then the whole expansion is well-formed. This will formally be demonstrated in section 3.3.

The above rules permit for example, the n-free frame buffer to be specified by NP1:

NP1 #i:1,n,1[path DEPOSIT(i);REMOVE(i) end]

and m pipelines of size n each associated with a mechanism controlling exits similar to that in the bounded delay priority queues in [LT79, C80] to be specified by:

```
NP2 #i:1,m,1
      [#j:1,n,1[path TR(i,j);TR(i,j+1) end]
       path TR(i,n+1);CS_END(i) end
      ]
```

We impose the restriction BRrest on bodyreplicators

(BRrest)

The range of the bodyreplicator indices should be non empty.

guaranteing that bodyreplicators generate at least one regularity. This is important, for a macro program body could consist of just bodyreplicators which upon expansion should generate a non-empty basic program body.

3.2.4 The Paths and Processes

Their syntax will be similar to the syntax of paths and processes of basic COSY:

MN9. mpath=path (msequence)* end
MN10. mprocess=process (msequence)* end

We have used "msequence" instead of "sequence" to stand for "macro sequence" since we will allow replicators and distributors as parts of them. Similarly, in the syntax rules below, "morelement" will stand for "macro orelement":

MN11. msequence={morelement @;}+
MN12. morelement={gelement @,}+
MN13. gelement=starelement/sreplicator/distributor
MN14. starelement=element/element*
MN15. element=operation/indexedop/(msequence)
MN16. operation=lc-letter{lc-letter/digit/_}*
MN17. indexedop=arrayid({iexpr @,}+)

In the above rules, "gelement" stands for "generalized element" since it

can be any of starelement, replicator or distributor. We have used "sreplicator" to indicate replicators which expand to basic COSY sequences which we call sequence replicators. The only difference between the above rules and corresponding ones in basic COSY is that here we allow three new types of elements, sequence replicators and distributors, produced by "sreplicator" and "distributor" respectively, which cannot be starred, and indexed operations, produced by "indexedop" . The above rules satisfy our third consideration for developing this grammar since it is structurally similar to the grammar of basic COSY. It is clear that any basic COSY program may be produced by the rules obtained up to now. According to these, a macro program could consist of just macro paths and processes the macro sequences of which do not involve any sequence replicators or distributors or indexed operations. But such a program could be produced by the basic COSY syntax as well. In addition "msequence" may involve any number of the three new types of elements.

We did not permit replicators and distributors to be starred as the star will not apply to the whole of the string they would generate, but only to its last element.

3.2.5 The Sequence Replicators

As we have noted in the previous section 3.1, the syntax rules for replicators in sequences produced replicators which are either too wide, not generating well-formed basic COSY strings when expanded, and meta-restrictions need to be applied, or are not general enough, generating a class of regularities which is not as large as we would like it to be. On the other hand we require that the replicators should be readable without formal expansion. For this reason we shall exclude replicators generating certain types of regularities. Before we give any syntax rules, let us specify exactly which replicators we will exclude. From the discussion in the section 3.1 it is obvious that we would like to avoid the production of some replicators, namely those which were well-formed only for a particular range of their index, the range dependent replicators. The path P40 for example

```
P40 path (([A(i);B(i))@,⌈i⌋|1,n,1] end
```

expands to a well-formed string only for n=2. We will require that when a replicator expands into a well-formed basic COSY string it does so for any non-empty range of its index.

Sometimes we may have a choice in generating a string. Should our replicators be so general as to be able to generate a string in any way or should they be more restricted? Is the shortest replicator always the "best"? To demonstrate the problem in deciding the "best" grammar let us consider the following example:

```
P70 path
      (A(1);B(1))
     ,(C(1);D(1),A(2);B(2))
     ,(C(2);D(2),A(3);B(3))
     ,(C(3);D(3))
    end
```

We may use replicators to abbreviate the regular substructures of the outermost orelement in P70. Two obviously similar substructures are the two middle elements of the orelement of P70, which may be generated using the old notation by:

```
P71 path
      (A(1);B(1))
     ,[(C(i);D(i),A(i+1);B(i+1))@,⌈i⌋|1,2,1]
     ,(C(3);D(3))
    end
```

But if we examine the orelement in P70 more carefully we see that another regular pattern is the string

```
A(i);B(i)),(C(i);D(i) for i=1,2,3.
```

So P70 may be generated by:

```
P72 path([A(i);B(i)),(C(i);D(i)@,⌈i⌋|1,3,1])end
```

Although P72 is the most concise path generating P70 it has some drawbacks. The regularity of the replicator cannot be described syntactically in terms of the non-terminals used in basic COSY because of the unmatched opening and closing parentheses. On the other hand the regularity in the replicator P71 may be described as:

(sequence)@,
(orelement;orelement;orelement)@,
(element;element,element;element)@,
etc.

Furthermore, the expansion of the replicator in P72 is only well-formed in the context "(...)" and not in any other context of any sequence replicator. We shall call this type of a replicator context dependent. As we would like our replicators to be well-formed when expanded in any possible context these replicators will not be permitted.

There is yet a third kind of a replicator we will not permit, the expansion of which does not depend on the separators on its left and on its right but on other replicators nearby. Consider for example the stack written as:

P73 path [(UP(i)@;|i|1,n,1];[DOWN(i))*@;|i|n,1,-1] end

which may be produced by the grammar of [LS77]. When the replicators in P73 are expanded, the resulting path is well-formed in basic COSY. We shall call these replicators neighbourhood dependent.

All three types of replicators we shall exclude have a common characteristic. They do not generate sequences themselves but only together with other parts of the macro sequence in which they are embedded. Since replicators may appear as non-starred elements in a macro sequence and according to our first consideration should produce well-formed basic COSY programs when expanded, they should generate basic COSY sequences. The replicators and the distributors according to rules MN9 to MN15 may only appear after any of "path", ",", ";", "(" and before any of "end", ",", ";", ")" and what may legally be written between any of these is a string generated from the non-terminal

"sequence" of basic COSY. The expansion of these replicators should always be non-empty, otherwise collision of terminal symbols will arise. We will obtain replicators which when expanded always generate well-formed basic COSY strings in any context they appear in the macro sequences. Here we do not try to produce replicators which generate syntactically strong strings (cf. section 3.1.5 grammar of [L79]). A part of a basic sequence which is a sequence itself is said to be syntactically strong in its context, if no parts of it bind with parts in its context. Our intention is to obtain a grammar which produces replicators generating a large class of well-formed basic COSY strings. In chapter 4 we shall give alternative rules for macro sequence by which all replicators in sequences generate syntactically strong strings. Although such replicators restrict the power for conciseness of macro programs, they have the advantages that it is not necessary for a macro sequence to be completely expanded for its semantics to be understood, and that they improve the readability of macro sequences significantly.

We would like to extend replicators to be able to generate imbrication of regularities which could not possibly be generated by any single replicator we examined in section 3.1, owing to the restrictive operational semantics of "@". These include sequences of path P43 for example

```
P43 path (A(1),(A(2),(A(3);B(3)),B(2)),B(1)) end
```

in which all regularities are of the form

```
p( ) s1...s1 q( )
```

except the innermost which is of the form

```
p( ) s2 q( )
```

where "s1" and "s2" are one of ";" or "," but not the same. We had pointed out in section 3.1.5 that to generate these kind of regularities the "@" should not only strip separators but replace them by others. We shall modify the replicator notation to deal with this extension and bring it into the same form as the rest of the replicators we have

developed up to now in this section. Let us first give the rules according to which a replicator in the old notation expanding to a basic COSY sequence is transformed into the new. The replicators which most of the grammars permit and which expand to sequences are of three types:

A. [p(i) @ sep|i|q(i)|in,fi,inc]
B. [p(i)|i|q(i) @ sep|in,fi,inc]
C. [p(i) @ sep|i|in,fi,inc]

where "p", "q" are strings which may involve subscripted operations the indices of which may depend on the replicator index "i". We shall transform A, B and C to the new notation in five simple steps. For each step we indicate to which type it will apply as some of the steps apply only to one or two types. When a step is applied the new intermediate forms of A, B, C will be given and will be identified by superscripting A, B, C by an integer denoting the number of transformations this type has undergone until that point. We assume that A, B, C are the same as A^0, B^0 and C^0 respectively. The five transformation steps are:

step1

applied to A^0: put after "|i|" the symbol "@".
applied to B^0: put before "|i|" the symbol "@".

A^1. [p(i) @ sep|i|@ q(i)|in,fi,inc]
B^1. [p(i) @|i|q(i) @ sep|in,fi,inc]

step2

applied to B^1: move "@ sep" immediately after "|i|".

B^2. [p(i) @|i|@ sep q(i)|in,fi,inc]

step3

applied to A^1,C^0: change "@ sep" to "sep @".

A^2. [p(i) sep @|i|@ q(i)|in,fi,inc]
C^1. [p(i) sep @|i|in,fi,inc]

step4

applied to A^2, B^2, C^1: since the two "@"s in A^2 and B^2 are sufficient to demarkate the patterns on the left and on the right of "[i]" we may remove the place holder, change the index specification to "#i:in,fi,inc" and move it in front of "[ ]" as we did for the other replicators. The same may be done for C^1.

A^3. #i:in,fi,inc[p(i) sep @ @ q(i)]
B^3. #i:in,fi,inc[p(i) @ @ sep q(i)]
C^2. #i:in,fi,inc[p(i) sep @] C is now in its final form.

step5

applied to A^3 and B^3: if q(i) in A^3 is of the form "sep1 q'(i)" and p(i) in B^3 of the form "p'(i) sep1" copy the separator "sep1" leading q(i) and respectively terminating p(i) between the two "@"s in A^3 and B^3 respectively.

A^4 #i:in,fi,inc[p(i) sep @ sep1 @sep1 q'(i)]
B^4 #i:in,fi,inc[p'(i) sep1 @ sep1 @ sep q(i)]

A and B are now in their final form. If q(i) and p(i) in A^3 and B^3 respectively are not in the appropriate form, step 5 is not applied and A^3 and B^3 are therefore in their final form.

Let us apply these transformations to three replicators R1, R2 and R3 in the old notation corresponding to the types A, B, C respectively. In the expressions below R1i, R2i and R3i will correspond respectively to forms Ai, Bi and Ci for i=0,...,4 of the above conversion rule.

The replicator R1

R1. [(SKIP(i)@;[i]),DO(i)|1,n,1]

after step 1 becomes $R1^1$

$R1^1$. [(SKIP(i)@;[i]@),DO(i)|1,n,1]

which after step 3 becomes $R1^2$

R1^2. [(SKIP(i);@ i @),DO(i)|1,n,1]

which finally after step 4 becomes R1^3

R1^3. #i:1,n,1[(SKIP(i);@ @),DO(i)]

As step 5 cannot be applied this is now in the new notation.

The replicator R2

R2. [(UP(i); i DOWN(i))*@;|1,n,1]

after step 1 becomes

R2^1. [(UP(i);@ i DOWN(i))*@;|1,n,1]

which after step 2 becomes R2^2

R2^2. [(UP(i);@ i @;DOWN(i))*|1,n,1]

which after step 4 becomes R2^3

R2^3. #i:1,n,1[(UP(i);@ @;DOWN(i))*]

taking its final form after step 5

R2^4. #i:1,n,1[(UP(i);@;@;DOWN(i))*]

The replicator R3

R3. [(A(i);B(i))@, i |1,n,1]

after step 1 becomes R3^1

R3^1. [(A(i);B(i)),@ i |1,n,1]

taking its final form after step 4

R3[2]. #i:1,n,1[(A(i);B(i)),@]

The replicators generating basic COSY sequences in the new notation are of two types:

the concatenator

generating consecutive regularities and are of the form

(Conc) #i:in,fi,inc[p(i) sep @] and

the imbricator

generating regularities nested within each other and are of the form

(Imbr) #i:in,fi,inc[p(i) @ t @ q(i)]

where "p", "t", "q" denote "patterns" and "sep" one of the separators ";" or ",". For concatenators and imbricators to expand to the same strings as replicators in the old notation of types C and A, B respectively, the operational semantics of "@" have to be changed.

In the concatenator the "@" strips the separator in front of it in the last copy of the regularity "p(i) sep". The expansion of the concatenator therefore looks like:

(concexp)

p(in) sep p(in+inc) sep ...sep p(fi')

where "fi'" denotes the final value of the range of the index which may be different from "fi".

In the imbricator the separators before the first "@" and after the second "@" in the last copy of the regularity "p( )q( )" will be replaced by "t". The expansion of the imbricator looks like:

(imbrexp)

p(in) p(in+inc)...p'(fi') t q'(fi')...q(in+inc) q(in)

In the above expression "fi'" is the same as in (concexp) and "p'", "q'" are the same as "p", "q" respectively but with any trailing separator of "p" and any leading separator of "q" respectively, removed.

The reason we have specified a string "t" to be between the two "@"s instead of just a separator is that we would like our grammar to permit paths such as NP3

NP3 path empty,#i:1,n,1[(UP(i);@;full*;@;DOWN(i))*] end

which specifies a stack of size n with tests for empty and full. When NP3 is expanded for n=3 for example the path NP4 is obtained:

```
NP4   path
          empty
         ,( UP(1)
           ;(UP(2);(UP(3);full*;DOWN(3))*;DOWN(2))*
           ;DOWN(1)
          )*
      end
```

in which the starred operation "full*" appears only once, in the innermost regularity. In general we shall permit any string to appear at that position as long as it forms a well-formed basic COSY string with the rest of the expansion.

Having specified what kind of replicators we will permit, and having decided on the notation of sequence replicators, we proceed to obtain their formal syntax rules. The approach we follow here is not to leave "p", "t", "q" as "patterns" but to specify more precisely what these may be syntactically. The syntax of the two types of replicators, concatenators and imbricators, shall be considered separately. The non-terminal "sreplicator" producing sequence replicators is defined as follows:

MN18. sreplicator=index_spec[{concseq/imbrseq}]

where the non-terminals "concseq" and "imbrseq" produce to the string

inside "[ ]" of concatenators and imbricators respectively. We next give the syntax for "concseq" and "imbrseq".

The non-terminal "concseq"

Before we give the syntax rules for the non-terminal "concseq", let us examine informally what p(i) in Conc should be syntactically. Its expansion (concexp) has been schematically given by:

p(in) sep p(in+inc) sep...sep p(fi')

For this string to be a well-formed basic COSY sequence each of "p( )" may in general be a sequence as we shall formally prove in 3.3.1. Therefore we may define "concseq" as

concseq=msequence sep @

which in principle is the syntax given in [L76, TL77].

According to the above rules though, ";" and "," in the context before "@" have equal precedence, whilst in the rest of the macro notation and in basic COSY "," has precedence over ";". To avoid this mixed precedence we shall consider the string produced by "concseq" as a regular expression with the symbol "@" appearing once as the last "element":

concseq={morelement;}* concor
concor={gelement,}* @

in which "concor" stands for the special "orelement" which contains as its last "element" the symbol "@". According to the above rules "," has precedence over ";" as in basic COSY. However, the above rules permit "concseq" to produce the string consisting of just "@", and therefore the replicator

#i:in,fi,inc[@]

may be produced which replicates the empty regularity thus generating an

empty expansion. To avoid the production of this empty replicator we finally define "concseq" as:

```
concseq={morelement;}* concor
       /{morelement;}+ @

concor={gelement,}+ @
```

The path $R3^3$ obtained from R3 by the conversion rule from the old to the new notation is permitted by the above rules. Also the replicators NR1 and NR2:

```
NR1 #i:1,n,1[DEPOSIT(i);@]
NR2 #i:1,n,1[#j:1,k,1[A(i,j),@];@]
```

In section 3.3.1 we shall formally prove that each of the replicators produced by these rules expand to macro sequences in general, and that their complete expansion forms a basic COSY sequence.

The non-terminal "imbrseq"

Before we give syntax rules for "imbrseq" let us examine informally what imbricators should generate so that when completely expanded they always generate well-formed basic COSY sequences. Let us first consider just the outermost regularity of their expansion (imbrexp):

```
p(in)...q(in)
```

Since (imbrexp) on the whole forms a sequence, p(in) must be a legal head of a sequence and should start with either an operation or "(". Similarly, q(in) must be a legal tail of a sequence and must terminate with an operation or ")" or ")*". Let us now examine the first and the second regularity of the expansion (imbrexp):

```
p(in) p(in+inc)...q(in+inc) q(in)
```

The strings "p(in)" and "p(in+inc)" must be legally connected if they together are to form a legal head of a sequence. Since these two

strings differ only in the integer expressions they involve, they start with the same symbols which implies that for the expansion of the imbricator (imbrexp) to be well-formed "p(i)" must terminate with ";" or "," or "(". Applying a similar argument to "q(in)" and "q(in+inc)" we determine that "q(i)" must start with ";" or "," or ")". Furthermore, the number of unmatched opening parentheses in "p(i)" should match with closing parentheses in "q(i)".

The above observations imply that the string "@ t @" appears in the string generated by "imbrseq" in the context in which generalized elements would appear. Therefore, the string generated by "imbrseq" may be considered as a "sequence" in which the string "@ t @" appears once only as a non-starred "element". In addition "t" may be of four different forms depending on its immediate enclosing context, that is depending on whether on the left of the first "@" is any of "[", "(" or a separator, and whether on the right of the second "@" is any of "]", ")" or a separator. If t is in the context

1. {;/,} @ t @ {;/,} then t=sep{/msequence sep}
2. {(/[} @ t @ {;/,} then t={/msequence sep}
3. {;/,} @ t @ {)/]} then t={/sep msequence}
4. {(/[} @ t @ {)/]} then t=msequence

where "sep" indicates one of ";" or "," and "msequence" a macro sequence.

Let us first give formal context-sensitive rules (CS) for "imbrseq":

(CS)

```
imbrseq={morelement ;}* imbror {; morelement}*
imbror={gelement ,}* imbrgel {, gelement}*
imbrgel=special_el/imbrstarel
imbrstarel=imbrel/imbrel*
imbrel=(imbrseq)
{;/,} special_el {;/,}={;/,} @ sep {/msequence sep} @ {;/,}
{(/[} special_el {;/,}={(/[} @ {/msequence sep} @ {;/,}
{;/,} special_el {)/]}={;/,} @ {/sep msequence} @ {)/]}
{(/[} special_el {)/]}={(/[} @ msequence @ {)/]}
```

The string "{s1/s2}" where s1 and s2 are one of ";", ",", "(", ")", "[" and "]" denote alternative equivalent contexts for "special_el". The symbols "[" and "]" are possible contexts for "special_el" in spite of the fact that they do not appear in the first five production rules of (CS) since the string produced by "imbrseq" is enclosed in "[ ]" (cf. MN18), and since "imbrseq" could just produce a string which "special_el" produces.

Let us apply the above rules CS to derive the strings inside "[ ]" of imbricators NR3, NR4 and NR5:

NR3 #i:1,n,1[(UP(i);RESET(i),@@)]
NR4 #i:1,n,1[(UP(i);@;@;DOWN(i))*]
NR5 #i:1,n,1[(SKIP(i);@@),V(i)]

The symbol "=>" in the derivations which follow, means that the leftmost non-terminal to the left of "=>" is replaced using a rule of the grammar to yield a string to the right of "=>". We shall use the symbol "=>+" to denote the derivation of a simple or indexed operation from a non-terminal, for brevity. For example the derivation

gelement=> starelement=> element=> indexedop=> UP(i)

may be abbreviated to

gelement=>+ UP(i)

The complete derivation of the string inside "[ ]" of NR3 is:

```
imbrseq =>  imbror => imbrgel => imbrstarel => imbrel =>
        =>  (imbrseq)
        =>  (morelement ; imbror)
        =>+ (UP(i) ; imbror)
        =>  (UP(i) ; gelement , imbrgel)
        =>+ (UP(i) ; RESET(i) , imbrgel)
        =>  (UP(i) ; RESET(i) , special_el)
        =>  (UP(i) ; RESET(i) , @@)
```

The complete derivation of the string inside "[ ]" of NR4 is:

```
imbrseq =>  imbror => imbrgel => imbrstarel => imbrel*
        =>  (imbrseq)*
        =>  (morelement ; imbror ; morelement)*
        =>+ (UP(i) ; imbror ; morelement)*
        =>  (UP(i) ; @ sep @ ; morelement)*
        =>  (UP(i) ; @ ; @ ; morelement)*
        =>+ (UP(i) ; @ ; @ ; DOWN(i))*
```

The complete derivation of the string inside "[ ]" of NR5 is:

```
imbrseq =>  imbror
        =>  imbrgel , gelement
        =>  imbrstarel , gelement
        =>  imbrel , gelement
        =>  (imbrseq) , gelement
        =>  (gelement ; imbror) , gelement
        =>+ (SKIP(i) ; imbror) , gelement
        =>  (SKIP(i) ; imbrgel) , gelement
        =>  (SKIP(i) ; special_el) , gelement
        =>  (SKIP(i) ; @@) , gelement
        =>+ (SKIP(i) ; @@) , V(i)
```

If t has its right form then when a replicator is expanded it will "bind" the left and the right expanded parts so that the resulting string may be produced by "sequence" of basic COSY.

The rules (CS) have two disadvantages: the syntax of "imbrseq" is given in terms of context-sensitive rules and the ";" and "," are of mixed precedence. Let us first obtain context-free rules keeping the mixed precedence of "," and ";". We will not express "@ t @" as an "element" on its own but together with other strings on its left and right so that the context of each of the four forms become distinguishable. The four forms may be expressed as:

1'. msequence sep @ sep{/msequence sep}@ sep msequence
2'. @ {/msequence sep} @ sep msequence
3'. msequence sep @ {/sep msequence} @
4'. @ msequence @

The context-free syntax rules for "imbrseq" in which ";" and "," have still mixed precedence are:

(CFm)

```
imbrseq={morelement ;}* imbror {; morelement}*
        /msequence sep @ sep{/msequence sep} @ sep msequence
        /@ {/msequence sep} @ sep msequence
        /msequence sep @ {/sep msequence} @
        /@ msequence @

imbror={gelement,}* imbrstarel{,gelement}*

imbrstarel=imbrel/imbrel*

imbrel=(imbrseq)
```

These syntax rules guarantee the production of well-formed macro COSY programs which when expanded produce well-formed basic COSY programs. This was possible by distinguishing the four different places where "@ t @" could appear. The string between the two "@" may contain, as it is clear from the syntax rules, a macro sequence. These syntax rules also allow any number of opening parentheses anywhere on the left of the first "@" and matching closing parentheses anywhere on the right of the second "@". Parentheses always match since they are produced in pairs.

Let us derive the strings inside "[ ]" of NR3, NR4 and NR5. First of NR3:

```
imbrseq =>  imbror => imbrstarel => imbrel
        =>  (imbrseq)
        =>  (msequence sep @ @)
        =>  (morelement ; morelement sep @ @)
        =>+ (UP(i) ; morelement sep @ @)
        =>+ (UP(i) ; RESET(i) sep @ @)
        =>  (UP(i) ; RESET(i) , @ @)
```

then of NR4:

```
imbrseq =>  imbror => imbrstarel => imbrel*
        =>  (imbrseq)*
        =>  (msequence sep @ sep @ sep msequence)*
        =>+ (UP(i) sep @ sep @ sep msequence)*
        =>  (UP(i) ; @ sep @ sep msequence)*
        =>  (UP(i) ; @ ; @ sep msequence)*
        =>  (UP(i) ; @ ; @ ; msequence)*
        =>+ (UP(i) ; @ ; @ ; DOWN(i))*
```

and finally of NR5:

```
    imbrseq =>  imbror
            =>  imbrstarel , gelement
            =>  imbrel , gelement
            =>  (imbrseq) , gelement
            =>  (msequence sep @ @) , gelement
            =>  (morelement sep @ @) , gelement
            =>+ (UP(i) sep @ @) , gelement
            =>  (UP(i) ; @ @) , gelement
            =>+ (UP(i)  ; @ @) , V(i)
```

In the syntax rules CFm however, the separators ";" and "," are of mixed precedence. The following context-free rules specify the precedence of "," over ";", but some meta-restriction rules are needed:

(CFr)

imbrseq={@/ } {imbror @;}+ {@/ }
imbror={imbrgel @,}+
imbrgel=imbrstarel/distributor/sreplicator/@
imbrstarel=imbrel/imbrel*
imbrel=operation/indexedop/(imbrseq)

The above rules do not constrain the production of "@". Any number of "@"s may appear in the strings produced by the above rules. Therefore, meta-restriction rules are needed to exclude certain strings:

(MR3)

(i) Only two "@" should be produced, and

(ii) the string "@...@" should be in its appropriate form according to its context.

The above syntax rules CFr and meta-restriction rule MR3 are based on a different approach from other syntax rules and corresponding meta-restrictions. Instead of leaving the symbols free, constraining them by meta-restrictions MR1 or MR2, we specify the "patterns" these symbols may form, leaving the number of "@"s free. Therefore, the checking of an imbricator for well-formedness is simplified, it being necessary only to check a substring, rather than the whole string, namely the substring "@ t @" and its immediate context.

We may derive the strings inside "[ ]" of the replicator NR3, NR4 and NR5, applying the syntax rules of CFr as follows. Firstly, of NR3:

imbrseq => imbror => imbrgel => imbrstarel
=> imbrel => (imbrseq)
=> (imbror;imbror @)
=>+ (UP(i);imbror @)
=> (UP(i);imbrgel,imbrgel @)
=>+ (UP(i);RESET(i),imbrgel @)
=> (UP(i);RESET(i),@ @)

then of NR4:

```
imbrseq =>  imbror => imbrgel => imbrstarel
        =>  imbrel*  => (imbrseq)*
        =>  (imbror ; imbror ; imbror ; imbror)*
        =>+ (UP(i) ; imbror ; imbror ; imbror)*
        =>  (UP(i) ; @ ; imbror ; imbror)*
        =>  (UP(i) ; @ ; @ ; imbror)*
        =>+ (UP(i) ; @ ; @ ; DOWN(i))*
```

and finally of NR5:

```
    imbrseq =>  imbror
            =>  imbrgel , imbrgel
            =>  imbrstarel , imbrgel
            =>  imbrel , imbgel
            =>  (imbrseq) , imbrgel
            =>  (imbror , imbror @) , imbrgel
            =>+ (SKIP(i) ; imbror @) , imbrgel
            =>  (SKIP(i) ; @ @) , imbrgel
            =>+ (SKIP(i) ; @ @) , V(i)
```

We would like however, to avoid the use of meta-restrictions altogether. To accomplish this we follow the approach in (CFm) letting the string inside the innermost "(...)" which contains "@ t @" to be produced by a non-terminal "imbr_at_seq". The rest of the string produced by "imbrseq" will look like an macro sequence. The non-terminal "imbrseq" may be defined by:

(CF)

imbrseq=imbr_at_seq
/{morelement ;}* imbror {; morelement}*

imbror={gelement ,}* imbrstarel {, gelement}*

imbrstarel=imbrel/imbrel*

imbrel=(imbrseq)

We have to specify what "imbr_at_seq" produces. We will consider it to produce one of the following strings each corresponding to one of the forms 1′ to 4′:

1". A regular expression in which the two "@" are included as special elements.

2". A regular expression including one "@" as a special element, headed by an "@".

3". A regular expression including one "@" as above, followed by an "@".

4". A regular expression headed and followed by "@"s.

We need to specify where the symbols "@" in these "regular" expressions are to appear. They may appear on their own as single non-starred elements. They may also appear in "orelements" as non-starred elements. We shall denote the "orelements" in which they are to appear by "at_or". Since these may contain one or two instances of "@" we suffix "at_or" by either "1" or "2". Furthermore, we need to specify where in "at_or" the "@"s are to appear. For "at_or1" we may distinguish three cases in which the "@" may be in front, in the middle, or at the back and for "at_or2" four cases in which the first "@" is in front and the second in the middle or the first in front and the second at the back, or the first in the middle and the second at the back, or both in the middle. Therefore we shall need seven non-terminals: "at_or1f", "at_or1m",

"at_or1b" and "at_or2fm", "at_or2fb", "at_or2mb", "at_or2mm" producing each of the "orelements" we described above. Their syntax is:

```
at_or1f=@ {,gelement}+
at_or1m={gelement ,}+ @ {, gelement}+
at_or1b={gelement ,}+ @
at_or2fb=@ {, gelement}* , @
at_or2fm=@ {, gelement}* , @ {, gelement}+
at_or2mb={gelement ,}+ @ {,gelement}* , @
at_or2mm={gelement ,}+ @ {,gelement}* , @ {, gelement}+
```

Let us give some examples of productions of the above non-terminals:

```
at_or1f  => @ , gelement =>+ @ , A(i)

at_or1m  => gelement,@,gelement =>+ A(i),@,gelement =>+ A(i),@,B(i)

at_or1b  => gelement,@ =>+ RESET(i),@

at_or2fb => @,gelement,@ =>+ @,ready,@

at_or2fm => @,@,gelement =>+ @,@,B(i)

at_or2mb => gelement,@,@ =>+ A(i),@,@

at_or2mm =>  gelement,@,@,gelement =>+ A(i),@,@,gelement
         =>+ A(i),@,@,B(i)
```

The non-terminal "imbr_at_seq" may now be defined by:

```
imbr_at_seq=
      {morelement ;}+ {@/at_or1f/at_or1m/at_or1b} {; morelement}*;
              {@/at_or1f/at_or1m/at_or1b} {; morelement}+

      /{morelement ;}+ {at_or1f/at_or1m/at_or1b} {; morelement}*;
              {at_or1f/at_or1m}

      /{at_or1m/at_or1b} {; morelement}*;
              {@/at_or1f/at_or1m/at_or1b} {; morelement}+

      /{at_or1m/at_or1b} {; morelement}*;
              {at_or1f/at_or1m}

      /{morelement ;}+ {at_or2fm/at_or2mm/at_or2mb} {; morelement}+

      /{morelement ;}+ {at_or2fm/at_or2mm}

      /{at_or2mm/at_or2mb} {; morelemnt}+

      /at_or2mm

      /@ {morelement ;}* {at_or1f/at_or1m}

      /@ {morelement ;}* {@/at_or1f/at_or1m/at_or1b} {;morelement}+

      /{at_or1m/at_or1b} {; morelement}* @

      /{morelement ;}+ {@/at_or1f/at_or1m/at_or1b} {;morelement}* @

      /@ msequence @
```

The above rules are certainly context-free, specify the precedence of "," over ";" and as we shall formally prove in the next section, always produce replicators which when expanded yield macro sequences in general. These rules were obtained by keeping the production rules in (CFm) which did not involve "@" and by expressing the strings which were produced by productions of (CFm) involving the "@" as a "regular expression" with special "orelements" containing the special element

"@". This was necessary for the elimination of the mixed precedence of the two separators. The first eight of the production rules of "imbr_at_seq" in (CF) correspond to the form 1', the next two to 2', the next two to 3' and the final one to 4'.

Let us derive the strings inside "[ ]" of replicators NR3, NR4 and NR5 from the above rules. First of NR3:

```
imbrseq =>  imbror => imbrstarel => imbrel
        =>  (imbrseq)
        =>  (imbr_at_seq)
        =>  (morelement ; at_orlb @)
        =>+ (UP(i) ; at_orlb @)
        =>  (UP(i) ; gelement , @ @)
        =>+ (UP(i) ; RESET(i) , @ @)
```

then of NR4:

```
imbrseq =>  imbror => imbrstarel => imbrel*
        =>  (imbrseq)*  => (imbr_at_seq)*
        =>  (morelement ; at_orlf ; at_orlf ; morelement)*
        =>+ (UP(i) ; at_orlf ; at_orlf ; morelement)*
        =>  (UP(i) ; @ ; at_orlf ; morelement)*
        =>  (UP(i) ; @ ; @ ; morelement)*
        =>+ (UP(i) ; @ ; @ ; DOWN(i))*
```

and finally of NR5:

```
imbrseq =>  imbror => imbrstar , gelement
        =>  imbrel , gelement
        =>  (imbrseq) , gelement
        =>  (imbr_at_seq) , gelement
        =>  (morelement ; at_orlf @) , gelement
        =>+ (SKIP(i) ; at_orlf @) , gelement
        =>  (SKIP(i) ; @ @) , gelement
        =>+ (SKIP(i) ; @ @) , V(i)
```

As we have indicated in the introduction of this subsection sequence

replicators should not expand to empty strings.

3.2.6 Some More Replicators

One criterion for the generality of a macro COSY notation would be whether macro programs in this notation may represent basic programs which have been represented by macro programs in other macro notations. Although quite a number of extensions have been introduced so far in the notation sequence replicators have a limitation: they should not expand to empty strings. A replicator may generate empty strings for two reasons, either because its regularity is the empty string, or because the values of "in", "fi", "inc" are such that the range of the index is empty. The former situation cannot occur in replicators produced by the grammar introduced so far since empty regularities are not permitted by the syntax rules. These replicators are not useful anyway. The latter situation is excluded by our meta-restrictions on "in", "fi" and "inc" imposed to avoid collision of terminal symbols.

The only place where a replicator would sometimes expand to the empty string and sometimes not, is encountered in the non-starving banker in [LT78] where the string

S1 (BNKRD(1)[;par;rap i |1,n+1,-1];

was nested inside three replicators one of which had "1" as its index, ranging from n+1 to 1 in steps of -1. Obviously, the replicator in the above string expands to non-empty when 1=n+1 and even then only one copy of ";par;rap" is generated. The string S1 in the style of the new notation for replicators would look like

S2 (BNKRD(1)#i:1,n+1,-1[;par;rap];

which would not be permitted by our rules in any macro COSY program. The context of the replicator is not the context of a "gelement" and the regularity in "[ ]" cannot be produced by "concseq" or "imbrseq". If however, S2 were rewritten as

S3 (BNKRD(1);#i:1,n+1,-1[par;rap;@];

it would be a well-formed substring in a macro COSY program. When the replicators in S3 and S1 are expanded to non-empty strings they generate the same basic COSY string. When however, 1<n+1 then the expansion of the replicator in S3 would would yield the empty string, and S3 would become:

S4 (BNKRD(1);;

which is not a well-formed basic COSY string because of the collision of the two semicolons. We will permit some special kind of replicators which may expand to empty strings. These replicators should generate well-formed basic COSY strings whether expanded to empty or not. They should conform with our primary consideration for well-formedness after expansion. To permit this kind of replicators we need to extend the notation and modify one of our syntax rules.

To avoid collision of separators when these replicators generate empty strings as in S4, their context should not be the same as that of generalized elements. A separator should be "missing" either on their left or on their right. Their expansion has to provide the extra separator. If the separator on their left is missing they will be called left replicators and will be produced by the non-terminal "lreplicator" and if the separator on their right is missing, they will be called right replicators and will be produced by the non-terminal "rreplicator". For their expansion to bind correctly, right replicators should precede and left replicators should follow starelements, sequence replicators and distributors. The syntax rule for "gelement", MN13 should be modified to

MN13. gelement={rreplicator}*
{starelement/sreplicator/distributor}
{lreplicator}*

The replicators produced by the non-terminals "lreplicator" and "rreplicator" will generate sequences with a separator preceding and following respectively. We shall define their syntax by the rules:

lreplicator=index_spec[{;/,}|{concseq/imbrseq}]
rreplicator=index_spec[{concseq/imbrseq}|{;/,}]

and therefore a replicator produced by the first rule will have the forms:

(Lconc) #i:in,fi,inc[sep|p(i) sep @]
(Limbr) #i:in,fi,inc[sep|p(i) @ t @ q(i)]

and by the second rule the forms:

(Rconc) #i:in,fi,inc[p(i) sep @|sep]
(Rimbr) #i:in,fi,inc[p(i) @ t @ q(i)|sep]

If their index range is empty the strings generated by their expansion will be empty as well. Otherwise the strings generated by the expansion of L(conc or imbr) and R(conc or imbr) will be the same as the strings generated by the expansion of the sequence replicator obtained from them by removing "sep|" and "|sep" respectively, preceded and respectively followed by "sep". In the above notation S3 would be written as:

NR6 (BNKRD(1)#i:1,n+1,-1[;|par;rap;@];

Although the replicator in the string S1 was used in [LT78] it cannot be produced by the grammar in that paper. As we noted in section 3.1.5 the grammar in [LT78] specified that replicators in sequences appear in the context of "elements". This kind of replicators may be produced only by the grammars which specified their regularities as strings together with MR2 or by the grammars in [LS80] and [SL80].

In certain cases the same basic COSY string could be generated by another replicator more economically than by a left or right replicator. Indeed the replicator

#i:1,n+1,-1[;par;rap]

generates the same string as the replicator in NR6 for any 1 and n. This is only possible when the separator before the "@" is the same as

the separator before "|" in left replicators or the separator after "|", which is not true in general. For example consider the left replicator:

NR7 #i:1,n,k[;|(A(i);B(i)),@]

and suppose it is nested within another replicator with index "k" ranging from 0 to n where n is a constant. For n=3 the possible expansions for NR7 would be:

for k=0: empty
for k=1: ";(A(1);B(1)),(A(2);B(2)),(A(3);B(3))"
for k=2: ";(A(1);B(1)),(A(3);B(3))"
for k=3: ";(A(1);B(1))"

No grammar for macro COSY given in the literature may produce replicators which generate the above strings at all, let alone more economically.

As we would like our replicators to have a fixed form we have chosen generality at the expense of some loss of conciseness rather choosing conciseness at the expense of generality.

3.2.7 The Distributors

As we have noted in section 3.1.9, the distributor able to generate the largest class of regularities was defined in [LSC81] by:

distributor=sep[msequence]

In the new notation we have replaced the round parentheses "(", and ")" around the string to be distributed by the square brackets "[" and "]" respectively to distinguish between basic COSY and macro COSY symbols.

What is inside "[ ]" is specified as a macro sequence. However, there is a difference between a macro sequence in a distributor and a macro sequence in paths and processes. The operations and indexed operations in the former are really array-slices. By an array-slice we

mean an equivalence class of indexed operations corresponding to the same collective name the indices of which differ in at least one dimension. Array-slices are represented like indexed operations but with the index fields, corresponding to the dimensions in which their elements differ, left blank. We call the dimensions corresponding to blank fields of an array slice the distributable dimensions of the array-slice. An array slice could have several distributable dimensions. When all the dimensions of an array slice are distributable then these define all the operations in the array and are represented by the collective name itself without any index fields at all. For example the collective names A and B defined by the collectivisor

NC7 array A(0:3) B(4,3) endarray

contain several slices. The collective name A has only one dimension and therefore only one array slice represented by

A( ) or A

defining the equivalence class of all the operations in A

[A(0),A(1),A(2),A(3)]

The collective name B has two dimensions and eight array slices:

B(1,)

defining the equivalence class

[B(1,1),B(1,2),B(1,3)]

and

B(2,), B(3,), B(4,), B(,1), B(,2), B(,3), B(,)=B

The only syntactic difference between macro sequence in distributors and macro sequence anywhere else is that some of the index fields of the

"operations" of the former may be empty . We suggest to define the "indexedop" in such a way that it would be syntactically valid for them to have some empty index fields. We can further restrict the "operations" involving blank fields to distributors by meta-restriction rules. The non-terminal "indexedop" will then be defined as:

MN17. indexedop= arrayid{({{iexpr/ }@ ,}+)/ }

Alternatively we could specify rules to distinguish the two macro sequences but this would almost double the number of our syntax rules.

A distributor operates on a specific distributable dimension of each array-slice in its macro sequence which after the expansion of the distributor ceases to be distributable. We shall refer to them as the distributable dimensions of a distributor. The array slices will be replaced upon the expansion of the distributor by sections of array-slices. By a section of an array slice we mean the equivalence subclass of operations in the array-slice which have the same index in one of the distributable dimensions of that slice. These sections can either be indexed operations or other array-slices with one distributable less dimension than the slice they originated from. For example slice A contains four sections which are indexed operations

A(0), A(1), A(2), A(3)

and slice B contains seven sections which are all array-slices

B(1,), B(2,), B(3,), B(4,), B(,1), B(,2), B(,3)

The distributable dimensions on which the distributor operates are said to be compatible when they all contain the same number of sections and the distributor is said to satisfy the compatibility criterion (CC1). Only if this compatibility criterion (CC1) is satisfied is the distributor well-formed and may be expanded. Before we specify how a distributor is expanded we need to define a total order on sections of distributable dimensions of array-slices. Since these sections differ from the others in the index value of one of their dimensions their order is natural to be defined according to these indices. The order of

the sections is defined to be the order in which these indices are generated in the array declarations. A distributor in which all the distributable dimensions on which it operates contain n sections may be expanded as follows:

> n copies of the macro sequence in the distributor will be concatenated separated by the separator associated with the distributor. In the first copy each array slice will be replaced by the first section in this slice. In general, the i'th copy ($1 \leq i \leq n$) of each array-slice will be replaced by the i'th section of this slice.

According to this scheme the distributor ND1

```
ND1 ;[A,B( ,3)]
```

where A, B are declared by NC7 expands to:

```
A(0),B(1,3);A(1),B(2,3);A(2),B(3,3);A(3),B(4,3)
```

The distributor implicitly introduces a total order on the sections of array slices, the order specified by the collectivisors. The order defined by the collectivisors is immaterial in a program without distributors. For example the substitution of NC7 by NC8

```
NC8 array #i:3,0,-1[A(i)] B(4,3) endarray
```

in a program MPROG would not affect at all the expansion of MPROG and therefore would not necessitate any changes to the rest of MPROG for its behaviour to remain unchanged, as long as, in general, A is not used in a distributor. If A were distributed then its expansion would depend on the collectivisor by which A was declared. With NC7 the order of the indices of its operations after the expansion will be ascending from left to right and with NC8 descending. However, we have to point out that although ,[A] produces different strings when A is defined by NC7 and NC8 this would not have any effect on the behaviour of MPROG, because of the semantics of "," and ";". The same of course is not

true for ;[A].

We shall extend the class of regularities which the distributors may generate by permitting them to distribute not only over the whole range of array-slices but over a subrange of them as well. We need to extend the notation for distributors to

ND2 sep #inind,fiind,incind[msequence]

in which "inind", "fiind", "incind" denote integer expressions, representing the subrange over which "msequence" is to be distributed. Using the subrange option we may restrict the expansion of a distributor to some selected "copies" of its regularity. The subrange defines which copies should be selected. The integer expressions "inind", "fiind", "incind" specify the first copy to be selected, the upper limit of the copies to be selected and the step by which the upper limit should be reached from "inind", respectively. Thus the copies to be selected in the expansion of ND2 are:

(inind)'th,(inind+incind)'th,...,(inind+(Ns-1)*incind)'th

where Ns is the number of copies to be selected.

For example the distributor ND3

ND3 ;#1,3,1[A]

would expand to:

A(0);A(1);A(2)

selecting out of all copies of "A" in the string generated by the expansion of ;[A] only the first, second and third. The distributor ND4

ND4 ;#1,3,2[A]

would expand to:

A(0);A(2)

selecting the first and the third copies of "A" in the expansion of ;[A].

We shall require the expansion of ND4 to be non-empty and the indices of the operations to be in the range defined by the collectivisors. In view of the semantics of the subrange the compatibility criterion (CC1) may be somewhat relaxed.

(Drest1)

When a subrange is defined the slices will not be required to contain the same number of sections but at least as many sections as specified by the subrange.

For example the distributor ND5

ND5 ;#2,3,1[B(1,),B(,1)]

where B is defined by NC6 or NC7 satisfies (Drest1) although B(1,) has three array slices and B(,1) four, and may be expanded to:

B(1,2),B(2,1)
;B(1,3),B(3,1)

A final point we have to examine is what interpretation will be given to the subrange when fiind<inind and incind<0 as in ND6

ND6 ;#3,1,-1[A]

There are three options:

1. to consider it as meaning the same as ND3 arguing that the subrange acts only as a selector and does not impose any order on these copies.

2. to consider it illegal arguing that it does specify an order which

nevertheless condradicts the order specified by ;[A].

3. to consider it as an extension of ;[A] and expand according to the subrange. The distributor ;[A] will be considered as an abbreviation for ;#1,4,1[A] in which no copies of A are excluded.

Of the three options only the third extends the power for abbreviation of the distributor, allowing more sequences to be generated, and for this reason we adopt it as the interpretation of the subrange. For example the sequence

A(3);A(2);A(1);A(0)

may be generated by the distributor ND7 by reversing the order of distribution of A:

ND7 ;#4,1,-1[A]

It is clear from the syntax of the distributors that these may be nested. Each of these distributors must apply to a different distributable dimension of each array-slice . The following restriction is imposed:

(Drest2)
Inside a k-nested distributor there must only be arrays with at least k dimensions out of which exactly k should be specified as their distributable dimensions.

Equivalently we may say that after the expansion of the outermost distributor, the rest of the distributors must obey the syntax rules. For example ND8

ND8 ;[,[A]]

where A is defined by NC7, is not valid since after the expansion of the outermost distributor a non-valid distributor is generated:

,[A(0)];,[A(1)];,[A(2)];,[A(3)]

The reason for this is that the macro sequences inside "[ ]" of the above expansion do not consist of array-slices but of operations.

We must specify which of the nested distributors applies to which of the distributable dimensions of array slices. The rule adopted in the past is that the outermost distributor will apply to the rightmost distributable dimension of each slice; the second outermost to the rightmost not allocated distributable dimension, etc. A possible relaxation of the above rule would be to consider it as the default rule and specify explicitly which separator applies to which distributable dimension. The distributor ND9 for example

ND9 ;[,[B]]

where B is defined by NC7 or NC8 would expand according to either rules to:

B(1,1),B(2,1),B(3,1),B(4,1)
;B(1,2),B(2,2),B(3,2),B(4,2)
;B(1,3),B(2,3),B(3,3),B(4,3)

with "," applying to the first dimension of B and ";" to the second. But ND10

ND10 ;1[,[B]]

would expand to :

B(1,1),B(1,2),B(1,3)
;B(2,1),B(2,2),B(2,3)
;B(3,1),B(3,2),B(3,3)
;B(4,1),B(4,2),B(4,3)

since it is explicitly specified that ";" applies to the first dimension of B and implicitly that "," applies to the rightmost unallocated dimension of B, according to the default rule.

The following restriction needs to be imposed on dimension selectors:

(Drest3)

The dimension selectors in distributors must have values [illegible] dimensions of array slices.

The complete syntax for the distributor would then be:

distributor={;/,}{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]

The feature for selection of distributable dimensions is very helpful when both ND9 and ND10 are required in the same program. Without it we had to use the equivalent replicator NR8

NR8 #i:1,4,1[,[B(i,)];@]

instead of ND10. If only one of ND9 or ND10 were required then we could define the collectivisor in such a way as to conform to the default rule. This extension is also important when distributing over dimensions of array slices in which the indices of the operations depend on some other dimension, like in NC8:

NC8 array #i:1,5,2[#j:1,i,1[S(i,j) T(j,i)]] endarray

where the indices in the second dimension of S depend on the indices of its first, and the indices of the first dimension of T on the indices in its sedond dimension. According to the expansion rules the distributor ND11

ND11 ;[,[T]]

expands to :

T(1,1)
;T(1,3),T(2,3),T(3,3)
;T(1,5),T(2,5),T(3,5),T(4,5),T(5,5).

However we cannot expand the nested distributor ND12

ND12 ;[,[S]]

since the second dimension of S must be distributed first and the number of operations in this dimension depends on the first. It will be required that when distributing over some dimension of collectivisors

which depend on other dimensions the indices of the latter must be known since otherwise the expansion is not defined.

However, our extension allows the distributor ND13

```
ND13 ;1[,[S]]
```

to be expanded instead of being obliged to write the replicator NR9:

```
NR9 #i:1,5,2[,[S(i, )];@]
```

Both ND13 and NR9 expand to:

```
 S(1,1)
;S(3,1),S(3,2),S(3,3)
;S(5,1),S(5,2),S(5,3),S(5,4),S(5,5)
```

To demonstrate the use of the two new features of distributors, the subrange and the facility of specifying distributable sections, let us consider two more "realistic" examples. In the first we shall specify the pipeline which, using just replicators may be written as

```
NP3 #i:1,n,1[path TRANSFER(i);TRANSFER(i+1) end]
```

where array TRANSFER is declared by

```
NC9 array TRANSFER(n+1) endarray
```

We may replace the sequence in the above path by a distributor obtaining

```
NP4 #i:1,n,1[path ;#i,i+1,1[TRANSFER] end]
```

In the second example we shall specify a square matrix which is initially empty. Processes may read or write to any element of the array asynchronously, but write's and read's on any element should alternate, and no read's should occur before the initial write. These

constraints may be specified by:

```
NC10 array WRITE READ(n,n) endarray
NP5  #i:1,n,1[#j:1,n,1[path WRITE(i,j);READ(i,j) end]]
```

A writer process which updates the elements of the matrix by columns may be specified by any of the following processes:

```
NP6 process #j:1,n,1[#i:1,n,1[WRITE(i,j);@];@] end
    process #j:1,n,1[;[WRITE(,j)];@] end
    process ;[#i:1,n,1[WRITE(i,);@]] end
    process ;[;[WRITE]] end
```

We now specify a number of processes each specifying reading from selected elements of the matrix. A process reading all the elements of the matrix by rows may be specified most concisely by

```
NP7 process ;1[;[READ]] end
```

A process reading the elements of the first r ($1 \leq r \leq n$) rows by columns may be specified by

```
NP8 process ;[;#1,r,1[READ]] end
```

A process reading the lower left triangular matrix may be specified by

```
NP9 process #i:1,n,1[#j:1,i,1[READ(i,j);@];@] end
```

or by

```
NP10 process #i:1,n,1[;#1,i,1[READ(i,)];@] end
```

Finally a process reading the elements of the matrix forming the upper right triangular matrix by rows may be specified by

```
NP11 process ;[#j:1,n,1[#k:j,n,1[READ(,k);@];@]] end
```

or by

```
NP12 process ;1[#j:1,n,1[;#j,n,1[READ];@]] end
```

We have now completed the development of the design and the syntax of the macro notation, except for the non-terminal "iexpr" producing integer expressions. The syntax rules for integer expressions may be found in appendix B together with the rest syntax rules for macro COSY. The syntax for "iexpr" in appendix B permits all integer expressions which have been used in macro programs.

The next section 3.3 is concerned with the expansion of replicators, distributors and of complete macro programs.

3.3 THE EXPANSION OF MACRO COSY PROGRAMS

In the last section 3.2 the expansion of replicators and distributors was given in a schematic way. In this section the expansion of replicators, distributors and of complete macro programs is formally defined. The strings obtained from their expansion are characterized. In particular macro programs are shown to expand to well-formed basic programs. We also prove a number of theorems for the replacement of macro elements in macro sequences by other macro elements generating the same strings as the former. In the three sub-sections of this section we examine the expansion of replicators, distributor and macro programs respectively.

3.3.1 The Expansion of Replicators

The replicators we developed in the previous section 3.2 are of the form

#index:in,fi,inc[s(index)]

where "index" is the replicator index, "in", "fi", "inc" are integer expressions and "s(index)" a string which may have various forms, depending on the type of replicator. If, for example, a replicator is a bodyreplicator then "s(index)" has the form:

p(index)

where p represents a collection of paths, processes and bodyreplicators the integer expressions in which may depend on "index". If a replicator is an imbricator then "s(index)" has the form:

p(index) @ t @ q(index)

where p and q are strings, the integer expressions in which may depend on "index" and t a string none of the integer expressions of which may depend on "index". For the purposes of this section we shall consider the general form of "s(index)" as being

(Gs) sep1| p(index) @ t @ q(index) | sep2

Of course none of the strings inside "[ ]" of any of our replicator has the general form (Gs), but all appropriate forms may be obtained from (Gs) by removing certain substrings. Therefore, a replicator may be considered as having the general form (GR)

(GR) #index:in,fi,inc [sep1 | p(index) @ t @ q(index) | sep2]

The parts of "s(index)" depending on "index", namely p(index) and q(index), may be repeated upon the expansion of replicators. The index specification part "#index:in,fi,inc" determines how many copies of these parts are to be made and the values the index takes which are to

order of the

be substituted in each copy for "index" upon the expansion of replicators. The values in the range of the index, if non-empty, form finite arithmetic progressions having initial value "in", difference "inc" and bound "fi". Under this interpratation of the index specification, the value for "inc" must be non-zero. Otherwise an infinite arithmetic progression would be formed with the value of "in" as the only element of the progression. If the number of copies to be generated is n (n>0) then the values the index takes are:

in,in+inc,in+2*inc,...,in+(n-1)*inc

The value of n is also determined by the index specification of replicators and is given by the formula:

n=(fi-in)//inc+1

where "//" denotes integer division. The above formula is well-defined since inc≠0. The value m=(fi-in)//inc gives the number of intervals of length |inc| from in to fi. If m is positive it indicates that fi may be approached from in in steps of inc and if negative that fi may be approached from in in steps of -inc. If m is zero it indicates that the distance from in to fi is less than |inc|. The index is to take values from in to fi in steps of inc. If m<0 then fi may not be aproached at all from in in steps of inc. In this case the index specification specifies an empty range for the values of index. If m>0 then fi may be approached from in in steps of inc and the values it may take are m+1. If m=0 then the index takes only one value, namely the value of in. Therefore, the index takes m+1 values and for a non-empty range

m+1=(fi-in)//inc+1=n>0

we have used the phrase "fi may be approached from in" instead of the phrase "fi may be reached from in" to indicate that fi acts as a bound not to be exceeded by index and does not necessarily specify the last value of index. For example the index specification

#i:1,6,2

specifies the values 1, 3, 5 for index, thus in that sence is equivalent to the index specification

#i:1,5,2

The values the replicator index may take may be generated by the formula:

(F) $f(j)=in+(j-1)*inc$ for $j=1,2,\ldots,n$.

When a replicator is expanded the values of f(j) for j=1,...,n will be substituted in the j'th copy of p(index) and q(index) for index.

Although the replicators generate various kinds of regularities produced by different syntax rules their expansion may be defined by one and the same formula. Let us first define the primitive-recursive operator COPY having three string arguments separated by "/":

$$\overset{1}{\underset{j=k}{\mathrm{COPY}}}\{P(j)/T/Q(j)\}= \begin{cases} \text{if } 1>k \text{ then } P(k) \overset{1}{\underset{j=k+1}{\mathrm{COPY}}} \{P(j)/T/Q(j)\}\ Q(k) \\ \text{if } 1=k \text{ then } P'(k)\ T\ Q'(k) \\ \text{if } 1<k \text{ then } T' \end{cases}$$

where P(j) and Q(j) are strings in which the integer expressions may depend on j. The strings P'(k) and Q'(k) are the same as P(k) and Q(k) respectively with the terminating and respectively leading separator,i removed. T is a string which does not involve integer expressions depending on j. The string T' is the same as T with both leading and trailing separators removed. Finally 1, k are integers.

The expansion of (GR) denoted by $\mathrm{replexp}^0(GR)$ will be given by the formula:

$$\mathrm{replexp}^0(GR)= \begin{cases} \text{if } inc\neq 0 \text{ and } n=(fi-in)//inc+1>0 \text{ or } t' \text{ non empty} \\ sep1\ \overset{n}{\underset{j=1}{\mathrm{COPY}}}\{p(f(j))/t/q(f(j))\}\ sep2 \\ \text{otherwise empty} \end{cases}$$

where p(f(j)) and q(f(j)) are obtained from p(index) and q(index)

respectively by substituting the function f(j)=in+(j-1)*inc for "index" and where the string t′ is the same as t with its leading and trailing separators removed. The superscript "0" in "replexp^0" indicates that only GR is expanded and not any other replicators which may be generated by its expansion.

Let us apply this formula to expand some replicators. In the expansion of the bodyreplicator NP12

NP12 #i:1,4,1[path DEPOSIT(i);REMOVE(i) end]

the regularity inside "[ ]" will be replicated (4-1)//1+1=4 times. In the symbolism of (GR)

p(index)="path DEPOSIT(i);REMOVE(i) end"

and t and q(index) are the empty strings. The expansion of NP12 denoted by replexp^0(NP12) is given by:

$$\underset{j=1}{\overset{4}{\text{COPY}}}\{\text{path DEPOSIT}(1+(j-1)*1);\text{REMOVE}(1+(j-1)*1)\text{end/ /}\}=$$

$$\underset{j=1}{\overset{4}{\text{COPY}}}\{\text{path DEPOSIT}(j);\text{REMOVE}(j)\ \text{end/ /}\}$$

which yields

path DEPOSIT(1);REMOVE(1) end
path DEPOSIT(2);REMOVE(2) end
path DEPOSIT(3);REMOVE(3) end
path DEPOSIT(4);REMOVE(4) end

Consider also the macro path NP13

NP13 path #i:1,4,2[DEPOSIT(i);@] end

specifying the sequentialization of the deposits in the odd frames of four free frame buffer specified by NP12. The expansion of the replicator in the macro sequence of the above path is given by the

formula:

$$\underset{j=1}{\overset{2}{COPY}}\{DEPOSIT(1+(j-1)*2;//\}=$$

$$\underset{j=1}{\overset{2}{COPY}}\{DEPOSIT(2*j-1);//\}=DEPOSIT(1);DEPOSIT(3)$$

The expansion of the imbricator in the path NP14

NP14 path empty,#i:1,k,1[(UP(i); @;full*;@ ;DOWN(i))*] end

when k=3 is given by:

$$\underset{j=1}{\overset{3}{COPY}}\{(UP(j);/;full*;/;DOWN(j))*\}$$

which yields

(UP(1);(UP(2);(UP(3);full*;DOWN(3))*;DOWN(2))*;DOWN(1))*

In the previous section we set the restriction that a sequence replicator should always expand to non empty strings. From the formula $replexp^0(GR)$ giving the expansion of GR it may be deduced that this restriction is formally expressed by:

(Rrest1)

inc≠0 and n=(fi-in)//inc+1>0 or t' non empty.

If k=0 in NP14 its expansion is still non empty and is given by

$$\underset{j=1}{\overset{0}{COPY}}\{(UP(j);/;full*;/;DOWN(j))*\}=full$$

Therefore NP14 after the expansion of its replicator for k=0 becomes:

NP15 path empty,full* end

which specifies that a "stack" of size 0 is both empty and full. If

however a stack may only be tested for empty as specified by NP16

NP16 path empty,#i:1,k,1[(UP(i); @;@ ;DOWN(i))* end

the expansion of the replicator in NP16 for k=0 is not defined, since (k-1)//1+1=0, the string t' is empty and consequently, the index specification does not satisfy (Rrest1). We may in this case use a left replicator, to which (Rrest1) does not apply, obtaining path NP17

NP17 path empty#i:1,k,1[,|(UP(i); @;@ ;DOWN(i))*] end

in which the replicator when k=0, yields the empty string and path NP18 is obtained

NP18 path empty end

which specifies that a stack of size 0 is always empty.

The condition n>0 also implies that the expression for n in Rrest is well-defined. If it is not then n>0 does not hold and the range of the replicator index is empty. Consider for example the two nested replicators

NR10 #j:0,2,1[#i:0,m mod j,1[A(i);@];@]

The index j of the outer replicator takes values 0, 1, 2. For j=0 the inner replicator becomes

NR11 #i:0,m mod 0,1[A(i);@]

As the expression "m mod 0" is not defined the range of i is empty and as the replicator is a concatenator does not satisfy (Rrest1).

The condition (Rrest1) for non-empty expansions is not the same as the one required in other notations. We may recall from the introduction of chapter 3 that the expansion of a replicator is empty when

inc=0 or (fi-in)*inc<0

the complement of which

$$inc \neq 0 \text{ and } (fi-in)*inc \geq 0$$

gives the condition a replicator expanding to non-empty strings. One obvious difference is that (Rrest1) could expand to non-empty when t' is non-empty irrespective of the values of in, fi, inc, as we demonstrated in the expansion of NP14 when k=0. But a more subtle difference is that the conditions

$$inc \neq 0 \text{ and } n=(fi-in)//inc+1>0 \qquad \text{(A) and}$$
$$inc \neq 0 \text{ and } (fi-in)*inc \geq 0 \qquad \text{(B)}$$

are not equivalent. Condition (B) certainly implies (A). Both require inc≠0. Condition (B) additionally requires that

$$\begin{aligned}(fi-in)*inc \geq 0 &\Rightarrow (fi-in)/inc \geq 0 \quad (inc \neq 0)\\ &\Rightarrow (fi-in)//inc \geq 0\\ &\Rightarrow (fi-in)//inc+1 \geq 1\end{aligned}$$

Therefore (B) implies (A). Let us now show that (A) does not imply (B).

$$\begin{aligned}n>0 &\Rightarrow (fi-in)//inc+1 \geq 1\\ &\Rightarrow (fi-in)/inc-e \geq 0 \quad (-1<e<1)\\ &\Rightarrow (fi-in)/inc \geq e\\ &\Rightarrow (fi-in)/inc>1 \quad \text{(as r.h.s. min. when e tends to -1)}\end{aligned}$$

For values of in, fi, inc satisfying

$$0>(fi-in)/inc>-1 \qquad \text{(I)}$$

that is

$$0>fi-in>-inc \quad \text{when } inc>0$$
$$0<fi-in<inc \quad \text{when } inc<0$$

also satisfy

(fi-in)*inc<0

For values of in, fi, inc satisfying (I) the number of values the index takes is

(fi-in)//inc+1=0+1=1

namely the value of in. Therefore for in, fi, inc satisfying (I) replicators in the new notation do expand to non-empty strings. Thus more replicators expand to non-empty strings under condition (A) than under (B). We have relaxed condition (B) for the folowing reasons. The value of fi is not always the last value the index takes. Thus we may replace fi by fi', the true final value index takes. We took the view that fi' is the integer closest to fi, such that (fi'-in) is an exact multiple of inc and fi' is either closer to in than fi or is the same as fi, as no integer in the range of an index could exceed fi. Mathematicaly fi' is defined by:

(i) (fi'-in) mod inc=0
(ii) |fi'-fi|<|inc|
(iii) |fi'-in|$\leq$|fi-in|

The value of fi' may be obtained by the formula:

fi'=in+(n-1)*inc

where n=(fi-in)//inc+1. When in an index specification

#i:in,fi,inc

fi is the true final value of index i both conditions (A) and (B) are equivalent as in, fi, inc cannot satisfy (I):

0>(fi-in)/inc>-1 (I)

as (fi-in) is an exact multiple of inc, thus

(fi-in)/inc=0

A final restriction has to be imposed on replicators not in collectivisors

(Rrest2)

The replicators should generate subscripted operations permitted by the collectivisors.

Replicators however may only generate some of the subscripted operations permitted by the collectivisors. In the expansion formula for replicators the index of COPY ranges from 1 to some integer n in steps of 1, no matter what the values of in, fi, inc are. This indicates that all replicators may be transformed to others the index specification of which has in=inc=1, expanding to the same string as the former. For example the replicator inside path NP19

```
NP19 path #j:1,2,1[DEPOSIT(2*j-1);@] end
```

has in=inc=1 and expands to the same string as the replicator inside path NP13

$$\overset{2}{\underset{j=1}{\text{COPY}}}\{\text{DEPOSIT}(2*j-1);//\}=\text{DEPOSIT}(1);\text{DEPOSIT}(3)$$

In fact there are families of replicators which all expand to the same string, differing in the index specification part and in the integer expressions inside "[ ]". The integer expressions inside "[ ]" of a replicator which may depend on the replicator index may be subscripting indexed operations, or may appear in index specifications of replicators and subrange specifications and dimension selection expressions. A replicator in which in=1 and inc=1 will be called the normal form of a replicator. We next prove two theorems, showing that all replicators may be replaced by replicators in normal form and that from replicators in normal form all replicators in the same family may be obtained. Let us first prove a lemma. The symbol "√√√" will indicate end of a proof.

LEMMA 1:

A string S obtained from a syntactic entity SE by replacing the integer expressions in SE by other integer expressions forms also the same syntactic entity.

Proof:

A syntactic entity is a string which may be produced by a non-terminal of the grammar. The string S may be produced by applying the same syntax rules as for producing SE down to the non-terminal "iexpr". Then the production for S diverges from that for SE in that different syntax rules may be applied to obtain integer expressions. Therefore S forms the same syntactic entity as SE.√√√

Let us now prove the theorem for the replacement of replicators by replicators in normal form.

THEOREM 3.1:

A replicator of the general form

(GR) #index:in,fi,inc[sep1 | p(index) @ t @ q(index) | sep2]

expands to the same string as the replicator in the normal form

(GR′) #j:1,n,1[sep1 | p(f(j)) @ t @ q(f(j)) | sep2]

where n=(fi-in)//inc+1 and f(j)=in+(j-1)*inc.

Proof:

As (GR) and (GR′) differ only in the integer expressions, by lemma 1, both may be produced by the same non-terminal. Consequently, (GR′) is a syntactically well-formed replicator.

The expansion of both (GR) and (GR′) is given by the same formula namely

$$\text{replexp}^0(\text{GR}) = \begin{cases} \text{if } inc \neq 0 \text{ and } n=(fi-in)//inc+1>0 \text{ or } t' \text{ non empty} \\ \text{sep1 } \underset{j=1}{\overset{n}{\text{COPY}}}\{p(f(j))/t/q(f(j))\} \text{ sep2} \\ \text{otherwise empty} \end{cases}$$

Therefore, (GR) may be replaced by (GR′).√√√

Let us now prove the theorem for replacement of replicator in normal form by general replicators.

THEOREM 3.2:

A replicator of the normal form

(GRNF) #j:1,m,1[sep1 | p(j) @ t @ q(j) | sep2]

expands to the same string as the replicator

(GR") #i:in,fi,inc[sep1 | p(g(i)) @ t @ q(g(i)) | sep2]

where in, inc are integers (inc≠0), fi=in+(m-1)*inc+e where

$0 \leq e < inc$ when inc>0 or
$0 \geq e > inc$ when inc<0

and g(i) the function g(i)=(i-in)//inc+1 and i does not appear in p,t,

Proof:

By lemma 1 the replicator (GR") is syntactically well-formed. The expansion of (GRNF) is given by

if m>0 or t′ is non-empty then

$$sep1 \ \underset{j=1}{\overset{m}{COPY}}\{p(i)/t/q(i)\} \ sep2$$

otherwise empty

The condition for a non-empty expansion for (GR") is

if inc≠0 and n=(fi-in)//inc+1>0 or t′ is non-empty

The value of inc is by definition non-zero. The value of n is given by:

n=(fi-in)//inc+1
=((in+(m-1)*inc+e)-in)//inc+1
=((m-1)*inc+e)//inc+1
=m-1+1 (as e//inc=0)
=m

Therefore the condition for non-empty expansion of (GR") is

if m>0 or t' is non-empty

The expression giving the expansion of (GR") is

if m≥1 or t' is non-empty then

sep1 $\overset{m}{\underset{j=1}{COPY}}${p(g(f(j))/t/q(g(f(j))} sep2

otherwise empty

where f(j)=in+(j-1)*inc. For the above expression to be the same as the expression for the expansion of (GRNF) the composite function g(f(j)) should be

g(f(j))=j

Let us demonstrate the validity of the above equality:

g(f(j))=(f(j)-in)//inc+1
=((in+(j-1)*inc)-in)//inc+1
=((j-1)*inc)//inc+1
=j-1+1
=j

Therefore the expansion of (GR") is the same as that of (GRNF).√√√

In the previous section we claimed that the syntax rules produce sequence replicators which when expanded generate macro sequences. Having formally defined the expansion of replicators we proceed in proving this claim. Without loss of generality we shall prove it for replicators in normal form which we assume have been produced by the non-terminals "sreplicator", "concseq" and by the (CF) rules for

"imbrseq".

Let us first prove some lemmata which we will use in proving our main theorems. From now on we assume without loss of generality that all replicators are in normal form.

LEMMA 2:

In a concatenator of the form

#i:1,n,1[p(i) sep @]

the string p(i) is a macro sequence.

Proof:

The string "p(i) sep @" is produced by the non-terminal "concseq" which produces in general, strings either of the form:

1. morelement1 ; morelement2 ; ... ; morelementn ; @

or of the form:

2. morelement1 ; ... ; morelementn ; gelement,...,gelement,@

If the substrings ";@" and ",@" are removed from 1 and 2 respectively the remaining strings correspond to p(i) and may be produced by the non-terminal "msequence". √√√

LEMMA 3:

If "s^1 sep s^2" is the string obtained by juxtaposition of two macro sequences s^1 and s^2 and the separator sep as shown, then it is a macro sequence.

Proof:

Let s^1 be:

morelement^{1}1 ; morelement^{1}2 ; ... ; morelement^{1}n

and s^2 be:

$$morelement^21 \ ; \ morelement^22 \ ; \ \ldots \ ; \ morelement^2m$$

If sep=; then "$s^1;s^2$" forms the macro sequence:

$$morelement^11 \ ; \ morelement^12 \ ; \ \ldots \ ; \ morelement^1n$$
$$; \ morelement^21 \ ; \ morelement^22 \ ; \ \ldots \ ; \ morelement^2m$$

If sep=, then the last macro orelement of s^1, namely "$morelement^1n$" and the first macro orelement of s^2, namely "$morelement^21$" in "s^1,s^2" form a macro orelement which we denote by "morelementc". Then the string "s^1,s^2" clearly forms the macro sequence:

$$morelement^11 \ ; \ morelement^12 \ ; \ \ldots \ ; \ morelement^1n-1$$
$$; \ morelementc$$
$$; \ morelement^22 \ ; \ morelement^23 \ ; \ \ldots \ ; \ morelement^2m$$

This completes the proof of this lemma.///

We may now prove our first theorem, the theorem for the expansion of concatenators to macro sequences:

THEOREM 3.3:

The expansion of a concatenator of the form of

(Conc) #i:1,n,1[p(i) sep @]

yields a macro sequence for any n>0.

Proof:

The expansion of the concatenator Conc is given by $replexp^0(Conc)$

$$\overset{n}{\underset{j=1}{COPY}}\{p(i)\ sep/\ /\}$$

To prove that this yields a macro sequence for any n>0 we shall use an inductive argument on n. When n=1 its expansion E(1) is given by p(1).

According to Lemma 2, p(i) is a macro sequence, and therefore according to Lemma 1 the string p(1) is a macro sequence, as well.

Assume that the expansion of Conc for a finite integer n, denoted by E(n) is a macro sequence which is of the form

E(n)=p(1)sep p(2)sep ...sep p(n)

The expansion of Conc for n+1 denoted by E(n+1) is given by:

E(n+1)=p(1)sep p(2)sep ...sep p(n)sep p(n+1)

which may be written as

E(n+1)=E(n) sep p(n+1)

According to lemmata 1 and 2 the string p(n+1) is a macro sequence. Therefore according to lemma 3 the string E(n+1) is a macro sequence since it is of the form "msequence sep msequence". By induction we deduce that Conc expands to macro sequences for any n>0.✓✓✓

Before we prove a similar result for imbricators let us prove three more lemmata.

LEMMA 4:

In an imbricator of the form

#i:1,n,1[p(i) @ t @ q(i)]

the string t′ obtained from t by removing its leading and trailing separators is either empty or a macro sequence in general.

Proof:

The string t is the part of the string produced by "imbr_at_seq" (cf. section 3.2) between the two "@"s. According to the first four production rules for "imbr_at_seq" the string t may be of one of the four forms:

1. ,gelement,...{; morelement}* ; gelement,...,
2. {; morelement}* ; gelement,...,
3. ,gelement,...;{morelement ;}*
4. ;{morelement ;}*

By removing the leading and the terminating separator the resulting string t′ may clearly be produced by "msequence" except in case 4 in which it may be empty.

According to the second group of four productions for "imbr_at_seq" the string t may be of two forms:

1. ,gelement,...,
2. ,

When the leading and terminating commas are removed from 1 the resulting string t′ is a macro orelement which certainly is a special case of a macro sequence. In the second case as t consists of just the "," when this comma is removed the resulting string t′ is the empty string.

According to production options 9 and 10 for "imbr_at_seq" the string t may be of the forms:

1. {morelement ;}* gelement,...,
2. {morelement ;}*

Clearly by removing the terminating separator from 1 and 2 either a macro sequence or an empty string is obtained.

According to production options 11 and 12 for "imbr_at_seq" the string t may be of the forms:

1. ,gelement,...{; morelement}*
2. {; morelement}*

Again, by removing the leading separator either a macro sequence or an empty string is obtained.

Finally, the last production option for "imbr_at_seq" specifies that t does not have leading or terminating separators and that it is a macro sequence.√√√

LEMMA 5:

If, in an imbricator of the form

#i:l,n,l[s]

where s is produced by "imbrseq", the two "@"s and the separators before the first and after the second "@"s are removed from s, then the resulting string is a macro sequence.

Proof:

The syntax rules (CF) for "imbrseq" show that it suffices to prove that if from a string sl produced by "imbr_at_seq" the two "@"s and some separators are removed as above then the resulting string is a macro sequence. The reason is that sl is either the complete string s or it appears as an element "(sl)" in s. The only difference between a macro sequence and a string produced by "imbrseq" is that the latter contains this special element. Therefore if sl after the above transformation becomes a macro sequence, the whole string s in "[ ]" will be one as well.

The first "@" in sl may appear after

(a1) "[" or "("
(b1) ";" or ","

Similarly the second "@" may appear before

(a2) "]" or ")"
(b2) ";" or ","

From the production rules for "imbrseq" it may be seen that all combinations of these contexts may occur. We shall consider each combination separately.

Case 1 : a1-a2

The first and the second "@" do not have any separator before and after them respectively. The string s1 is produced by the last production option for "imbr_at_seq". When the two "@" are removed from s1 the remaining string is a macro sequence.

Case 2 : a1-b2

The string s1 is produced by the 9th and 10th production options for "imbr_at_seq". The first "@" does not have any separator in front of it. When the first "@" is removed from s1 the resulting string s1' is like a macro sequence except for one of its "orelements" which involves "@" either on its own or in an orelement produced by one of "at_or1f", "at_or1m", "at_or1b". The string s1' may be of four forms:

1. {morelement ;}* @,gelement,...,gelement {; morelement}*
2. {morelement ;}* @ {; morelement}+
3. {morelement ;}* gelement,...,@,...,gelement {; morelement}*
4. {morelement ;}* gelement,...,@ {; morelement}+

When "@," and "@;" are removed from 1, 2 and 3 respectively the remaining strings are macro sequences. When "@;" is removed from 4 the string "gelement,...," together with the first macro orelement after "@;" is a macro orelement and therefore the whole of the remaining string is a macro sequence.

Case 3 : b1-a2

The string s1 is produced by the 11th and 12th production options for "imbr_at_seq". The second "@" is not followed by a separator. When the second "@" is removed from s1 the resulting string s1' is like a macro sequence except that one of its "orelements" involves the "@" either on its own or in an orelement produced by one of "at_or1f", "at_or1m", "at_or1b". The string s1' may be of the forms:

1. {morelement ;}* gelement,...,@ {; morelement}*
2. {morelement ;}+ @ {; morelement}*

3. {morelement ;}* gelement,...,@,gelement {; morelement}*
4. {morelement ;}+ @,gelement,...,gelement {; morelement}*

As in the previous case, when ",@" and ";@" are removed from 1, 2 and 3 respectively the remaining string is a macro sequence. When ";@" is removed from 4 the macro orelement in front of it together with ",gelement,..." is a macro orelement and therefore the whole of this string is a macro sequence.

Case 4 : b1-b2

This case has two subcases: Either the two "@"s are in two separate special orelements either on their own or in orelements produced by one of "at_or1f", "at_or_1m", "at_or_1b" when productions 1 to 4 for "imbr_at_seq" are applied, or both "@"s appear in the same special orelement produced by one of "at_or2fm", "at_or2fb", "at_or2mm", "at_or2mb" when productions 5 to 8 for "imbr_at_seq" are applied.

In the first subcase we apply the same arguments as in cases 2 and 3.

In the second subcase s1 may take four forms:

1. {morelement;}* gelement,...,@,...,@,...,gelement {;morelement}*
2. {morelement;}+ @,gelement,...,@,...,gelement {;morelement}*
3. {morelement;}* gelement,...,@,....,gelement,@ {;morelement}+
4. {morelement;}+ @,gelement,...,gelement,@ {;morelement}+

When ",@" and "@," are removed from 1 the resulting string is a macro sequence. When ";@" and "@," are removed from 2 the macro orelement before ";@" together with ",gelement,...,gelement" is a macro orelement and therefore the whole string is a macro sequence. Similarly, when ",@" and "@;" are removed from 3 the string "gelement,..." is a macro orelement and therefore the whole string is a macro sequence. Finally, when ";@" and "@;" are removed from 4 the macro orelement in front of ";@" together with the string ",gelement,...,gelement" and the macro orelement after "@;" is a macro orelement and the whole of that string is a macro sequence.√√√

LEMMA 6:

If in an imbricator of the form

(Imbr) #i:1,n,1[p(i) @ t @ q(i)]

the string t is replaced by a string t1 consisting of a macro sequence MSEQ preceded by the trailing separator of p and followed by the separator leading q, then the imbricator obtained may be produced by the syntax rules for "sreplicator" and the (CF) rules for "imbrseq".

Proof:

Let RS be the part of the string "p(i) @ t @ q(i)" produced by the non-terminal "imbr_at_seq". Since t appears only in this string it suffices to prove that the string obtained from RS by replacing t1 for t may be produced by "imbr_at_seq". Depending on whether p(i) terminates with and q(i) starts with a separator or not, the string t1 may be of four different forms. We shall consider each case separately.

Case 1 :p(i) does not terminate and q(i) does not start with a separator.

Then RS is of the form

@ t @

in which case t1 is of the form

t1=morelement1 ; ... ; morelementn

The string obtained by replacing t1 for t in RS may be produced by the last production option for "imbr_at_seq".

Case 2 :p(i) terminates but q(i) does not start with a separator.

The string RS may be of two forms. The first one is:

1. {morelement ;}* gelement,...,@ t @

in which case t1 is of the form

t1=,morelement1 ; ... ; morelementn

By replacing t for t1 in 1 the string

{morelement;}* gelement,...,@,morelement1 ; ... ; morelementn @

is obtained in which the substring

gelement,...,@,morelement1

may be produced by "at_or1m" and therefore the whole string may be produced by the 11th and 12th production options for "imbr_at_seq".

The string RS may also be of the form:

2. {morelement ;}+ @ t @

in which case t1 is of the form

t1= ; morelement1 ; ... ; morelementn

By replacing t1 for t in 2 the string

{morelement ;}+@ ; morelement1 ; ... ; morelementn @

is obtained which may be produced by the 12th production for "imbr_at_seq".

Case 3 :q(i) starts but p(i) does not terminate with a separator.

Again RS may be of two forms. The first one is:

1. @ t @,gelement,...,gelement {; morelement}*

in which case t1 is of the form

t1=morelement1 ; ... ; morelementn,

When t1 is replaced for t in 1 the substring of t1 "morelementn", together with "@,gelement,..." may be produced by "at_or1m" and the whole of the string by the 9th and 10th options for "imbr_at_seq".

The second form RS may take is

2. @ t @{;morelement}+

in which case t1 is of the form

t1=morelement ; ... ; morelementn;

When t1 is replaced for t in 2 the whole string may be produced by the 10th option for "imbr_at_seq".

Case 4 : p(i) terminates and q(i) start with a separator.

In this case RS may take four forms. The first one is:

1. {morelement ;}* gelement,...,@,...,@,...,gelement {;morelement}*

in which case t1 is of the form

t1=,morelement1 ; ... ; morelementn,

When t1 is replaced for t in 1 the string "gelement,...,@" together with ",morelement1" may be produced by "at_or1m". Similarly, "morelementn," together with "@,...,gelement" may be produced by "at_or1m". Therefore the whole string may be produced by one of the production options 1 to 4 for "imbr_at_seq", depending on whether "{morelement ;}*" and "{; morelement}*" in 1 represent at least one macro orelement or the empty strings.

In the special case where MSEQ="morelementl" then tl=",morelementl," and the string "gelement,...,@" together with ",morelementl," and "@,...,gelement" may be produced by "at_or2mm" and the whole string by production options 5 to 8 for "imbr_at_seq".

The second form RS may take is

2. {morelement;}+ @ t @,gelement,...,gelement {;morelement}*

in which case tl is of the form

tl=;morelementl ; ... ; morelementn,

When tl is replaced for t in 2 the string "morelementn," concatenated with "@,gelement,...,gelement" may be produced by "at_or2mm" and the whole string by the 6'th and 7'th options of "imbr_at_seq".

The third form RS may take is

3. {morelement;}* gelement,..., @t @ {;morelement}+

in which case tl is of the form

tl=,morelement ; ... ; morelementn;

By applying similar arguments as before the string obtained by replacing tl for t may be produced by productions 1 or 3 for "imbr_at_seq".

Finally, RS may take the form:

4. {morelement ;}+ @ t @ {;morelement}+

in which case tl is of the form

tl=;morelementl ; ... ; morelementn;

By similar arguments we may show that by replacing tl for t in 4 the resulting string may be produced by production option 1 of

"imbr_at_seq".✓✓✓

The theorem analogous to 3.3 may now be proven, the theorem for the expansion of imbricators to macro sequences:

THEOREM 3.4:

The string obtained by the expansion of an imbricator

(Imbr) #i:1,n,1[p(i) @ t @ q(i)]

is a macro sequence.

Proof:

The expansion of a replicator is valid when

$n \geq 1$ or t′ is not the empty string

We shall distinguish two cases:

1. $n<1$ and t′ is non empty.

Its expansion then is given by t′ which by lemma 4 is a macro sequence.

2. The second case is when $n \geq 1$.

To prove that the imbricator expands to a macro sequence we shall use an inductive argument for n. When n=1 the expansion of the imbricator denoted by E(1), is given by

$$E(1)=\overset{1}{\underset{i=1}{C}}OPY\{p(i)/t/q(i)\}$$

which yields

$$E(1)=p'(1)\ t\ q'(1)$$

which according to lemmata 5 and 1 is a macro sequence.

Assume that the string generated by the expansion of the imbicator, for some $n \geq 1$, denoted by E(n) is a macro sequence. Its expansion E(n) is given by

$$E(n)=\overset{n}{\underset{i=1}{COPY}}\{p(i)/t/q(i)\}$$

which yields

$$E(n)=p(1)\ p(2)...p'(n)\ t\ q'(n)...q(2)\ q(1)$$

Consider now the expansion E(n+1) given by

$$E(n+1)=\overset{n+1}{\underset{i=1}{COPY}}\{p(i)/t/q(i)\}$$

which yields

$$E(n+1)=p(1)\ p(2)...p(n)\ p'(n+1)\ t\ q'(n+1)\ q(n)...q(2)\ q(1)$$

If E(n) is a sequence then by lemma 1 so must E'(n)

$$E'(n)=p(2)\ p(3)...p'(n+1)\ t\ q'(n+1)...q(3)\ q(2)$$

obtained from E(n) by replacing the integer expressions of the subscripted operations and index specifications depending on "i" by the same expressions depending on "i+1" for i=1,2,...,n.

We now construct the imbricator

$$(R)\ k{:}1,1,1[p(k)\ @\ t1\ @\ q(k)]$$

where t1 is obtained from E'(n) prefixed by the terminating separator of p(i) and postfixed by the leading separator of q(i). According to lemma 6 this replicator is syntactically well-formed and its expansion E(R) is

$$E(R)=p'(1)\ t1\ q'(1)$$

which according to lemmata 1 and 4 is a macro sequence. Therefore E(R) is the macro sequence:

p(1) p(2) p(3)...p'(n+1) t q'(n+1)...q(3) q(2) q(1)

as the leading separator of tl together with p'(1) form p(1), the terminating separator of tl together with q'(1) form q(1) and the rest of tl is E(n). As E(R) is the same as E(n+1), the latter is a macro sequence.√√√

A similar result for generalized elements is the the theorem for the expansion of generalized elements to macro sequences. A generalized element GEL may be represented by

RLs M LLs

where LLs denote left replicators LR1...LRm and RLs right replicators RR1...RRn and M a sequence replicator or a distributor or a starelement. The expansion of all the left and right replicators of GEL will be given by $gelexp^0(GEL)$ defined as follows:

$gelexp^0(GEL)=a\ M\ b$

where $a= replexp^0(RR1)...replexp^0(RRn)$

$b= replexp^0(LR1)...replexp^0(LRm)$

We may now prove the theorem for the expansion of generalized elements to macro sequences.

THEOREM 3.5:

When all left and right replicators in a generalized element GEL are expanded the resulting string $gelexp^0(GEL)$ is a macro sequence.

Proof:

A generalized element GEL has the form

RR1...RRn {starelement/sreplicator/distributor} LR1...LRm

where RRi for i=1,...,n denote right replicators and LRi for i=1,...,m denote left replicators. From the definition of the expansion of left and right replicators and from theorems 3.3 and 3.4 it follows that the expansion of a right replicator is of the form

msequence sep or empty

and of a left replicator

sep msequence or empty

When all left and right replicators are expanded the generated string will be of the form:

{msequence sep}* {starelement/sreplicator/distributor} {sep msequence}*

Applying lemma 3 in the above string from left to right we deduce that this is a macro sequence.√√√

The expansion rule for replicators may be applied to expand any replicators of the general form

#i:1,n,1[p(i) @ t @ q(i)]

in which "@ t @ q(i)" and "t @ q(i)" may not exist and the string inside "[ ]" may not necessarily have been produced by "concseq" or "imbrseq". Let us call these replicators, wide replicators which may be of two forms: wide concatenators or wide imbricators. We may now prove the replacement theorem of imbricators by wide concatenators.

THEOREM 3.6:

Wide concatenators of the form

(Wconc) #i:1,n,1[s(i)@]

are sufficient to generate strings generated by imbricators of the

type

(Imbr) #j:1,n,1[p(j) @ t @ q(j)]

Proof:

Let us transform (Imbr) to the replicator form (RF)

(RF) #i:1,n,1[p(i)@] t #i:n,1,-1[q^1(i)@]

where q^1(i) is obtained by

$q^1(i)=$ if q(i)=sep q'(i) then q'(i) sep
else q(i)

The expansion of the replicators in (RF) is given by:

$\overset{n}{\underset{i=1}{COPY}}${p(i)/ /} t $\overset{n}{\underset{i=1}{COPY}}${$q^1$(n-i)/ /}

which yields E(RF)

E(RF)=p(1) p(2)...p'(n) t q^1(n)...q^1(2) $q^{1\prime}$(1)

The expansion of (Imbr) is given by:

$\overset{n}{\underset{i=1}{COPY}}${p(i)/t/q(i)}

which yields the string E(Imbr)

E(Imbr)=p(1) p(2)...p'(n) t q'(n)...q(2) q(1)

We will show that the strings E(RF) and E(Imbr) are the same.

They certainly have the same head "p(1)p(2)...p'(n)t". Therefore it suffices to show that the string (s1)

(s1) q^1(n)...q^1(2) $q^{1\prime}$(1)

is the same as the string (s2)

(s2) q'(n)...q(2) q(1)

If q(i) does not start with a separator (s1) is the same with (s2) for then q'(n) is the same as $q^1(n)$, q(1) the same as $q^{1\prime}(1)$ and each of q(j) is the same as $q^1(j)$ for j=2,3,...,(n-1).

Consider now the case where q(j) starts with a separator i.e. it is of the form

sep q'(j)

and therefore

$q^1(j)$=q'(j) sep

Substituting the right hand expression for q^1 in (s1) we obtain (s1')

(s1') q'(n)sep...q'(2)sep q'(1)

But each of "sep q'(j)" is the same as "q(j)" for j=1,2,...,(n-1). Substituting these expressions in (s1') the string

q'(n) q(n-1)...q(2)q(1)

is obtained which is the same as (s2).√√√

The next theorem characterizes the imbricators which when replaced by the transformation (RF) of the previous theorem the well-formedness of the macro programs is preserved. Let us first prove a lemma.

LEMMA 7:

If a generalized element GEL in a macro sequence MS is replaced by a macro sequence MSEQ the resulting string is a macro sequence.

Proof:

Without loss of generality we may assume that the element GEL is not nested inside "( )". For if it is we may consider the sequence MS' in the innermost element "(MS') which involves GEL. If by replacing MSEQ for GEL in MS' a macro sequence MSEQ' is obtained then "(MSEQ')" would be an element and the whole string a macro sequence. The sequence MS may be of the form

MS^1 GEL MS^2

where MS^1 may be either empty or in one of the forms

1. morelement^{1}1;morelement^{1}2 ; ... ; morelement^{1}n;

2. morelement^{1}1 ; ... ; morelement^{1}n;gelement,...,gelement,

and MS^2 may be either empty or in one of the forms

1. ;morelement^{2}1;morelement^{2}2 ; ... ; morelement^{2}n

2. ,gelement,...,gelement;morelement^{2}1 ; ... ; morelement^{2}n

If MS^1 and MS^2 are non empty it can be seen from their respective forms 1 and 2 that they are of the forms

msequence^1 sep and
sep msequence^2

respectively. Therefore MS may be of the following forms

1. msequence^1 sep MSEQ sep msequence^2
2. msequence^1 sep MSEQ
3. MSEQ sep msequence^2
4. MSEQ

Applying lemma 3 twice in 1 and once in each of 2 and 3 we prove that the forms in 1, 2, 3 are macro sequences. MSEQ in 4 is already a macro

sequence.///

Let us now prove a theorem giving the conditions for the replacement of imbricators by concatenators

THEOREM 3.7:

The imbricators of the form

(Imbr) #i:1,n,1[p(i) @ t @ q(i)]

may be equivalently replaced by the concatenator forms

(CF1) #i:1,n,1[p(i)@] t #i:n,1,-1[q^1(i)@]

when p and q are non empty, or by

(CF2) #i:1,n,1[p(i)@] t

when p is non empty but q is empty, or by

(CF3) t #i:n,1,-1[q^1(i)@]

when p is empty but q is not, or finally by

(CF4) t

when both p and q are empty,
where q^1 is obtained from q by transfering its leading separator to its back,
if and only if
p and q in (Imbr) do not contain any unmatched opening and closing parentheses respectively.

Proof:

(if)

Since p and q do not contain any opening and respectively closing

unmatched parentheses "p(i) @ t @ q(i)" is produced by "imbr_at_seq". We have to show that the replicators in the strings (CFi) for i=1,2,3,4 may be produced by "concseq" and that when the whole of (CFi) for i=1,2,3,4 is replaced for (Imbr) the well-formedeness of the program is preserved. We shall consider each of the four cases separately.

Case 1 : p and q are non empty.

From the production options 1 to 8 for "imbr_at_seq" which generate such strings it may be seen that p in general is of the form

msequence sep

the string t of the forms

sep msequence $^{sep}_{or}$ sep

and q of the form

sep msequence

which implies that q^1 is of the form

msequence sep

The strings p and q^1 appended by "@" may be produced by the syntax rule for "concseq":

```
concseq={morelement ;}+ @
        /{morelement ;}* concor
```

depending on whether p and q^1 terminate with ";" or ",". Therefore both replicators in (CF1) are legal. The whole string (CF1) is a macro sequence by lemma3. Replacing (Imbr) for (CF1) in a macro sequence the new string by lemma 7 is a macro sequence as well.

Case 2 : p is non empty, q is empty

From the production options 11, 12 which produce such strings it may be seen that p is of the form

msequence sep

and t is either empty or of the form

sep msequence

Using similar arguments to those used in case 1 we may show that the replicator in (CF2) is legal. By lemma 3 the whole of (CF2) is a macro sequence, and by lemma 7, by replacing it in a macro sequence for (Imbr) a macro sequence is obtained.

Case 3 : p is empty, q is non-empty.

From the production options 9, 10 for "imbr_at_seq" which produce such strings it may be seen that t is either empty or of the form

msequence sep

and q of the form

sep msequence

which implies that q^1 is of the form

msequence sep

Using similar arguments as in cases 1 and 2 we may prove that the replicator in (CF3) is legal and that when (CF3) is replaced for (Imbr) a macro sequence is obtained.

Case 4 : p and q are both empty.

From the last production of "imbr_at_seq" which generates such

strings we deduce that t is of the form

msequence

By lemma 7, if t or (CF4) is replaced for (Imbr) in a macro sequence then the new string is a macro sequence also.

From theorem 3.6 it follows that in each case (CFi) for i=1,2,3,4 expands to the same string as (Imbr).

(only if)

For every unmatched opening bracket in p there is an unmatched closing bracket in q and vice-versa. By lemma 2 in a replicator

#i:1,n,1[p(i) @]

the string p(i) is of the form

msequence sep

In a macro sequence of course all parentheses are matched. Therefore for the replicators in (CFi) for i=1,2,3 to be legal, p and q must not contain any unmatched opening and respectively closing parentheses.

The previous theorem gave the conditions under which an imbricator may be substituted by expressions involving concatenators. Obviously if t and one of p or q in (Imbr) are empty then (Imbr) may be replaced by a single concatenator. Under restricted conditions it it also possible to replace an imbricator in which possibly all three of p, t, q are nonempty by a single concatenator both expanding to the same string. The following theorem 3.7' for the replacement of an imbricator by a single concatenator gives these conditions.

THEOREM 3.7':

A well-formed imbricator

#i:1,n,1[p @ t @ q]

may be replaced by a single concatenator if and only if 1. any trailing separator of p, leading separator of q and trailing and leading separator of t must be the same.

2. the string p is either empty or of the form "p' sep", where p' may be generated by a concatenator of the form

j:1,k1,1[s(g1(i,j)) sep @]

the string q is either empty or of the form "sep q'", where q' may be generated by a concatenator of the form

#j:1,k2,1[s(g2(i,j)) sep @]

and the string t' obtained by stripping t of its leading and trailing separators may be empty or may be generated by a concatenator of the form

j:1,k3,1[s(g3(i,j)) sep @]

where k1, k2, k3 do not depend on i, and where s(g1(i,j)), s(g2(i,j)), s(g3(i,j)) denote that the indexed operations in s depend on expressions g1(i,j), g2(i,j), g3(i,j) respectively, which have the forms:

k1*(i-1)+j

n*k1+k3+k2*(n-i)+j

n*k1+j

respectively. If any of p, q, t' are empty then the corresponding values of k1, k2, k3 in the corresponding concatenators is taken as zero.

Proof (sketch):

(if)

We construct the concatenator

#i:1,m,1[s(i) sep @]

where m=n*(k1+k2)+k3. the values generated by
g1(i,j) for i=1,...,n and j=1,...,k1
g2(i,j) for i=1,...,n and j=1,...,k2
g3(i,j) for i=1,...,n and j=1,...,k3
are precisely the values 1,...,m=n*(k1+k2)+k3.

(only if)

1. the separators must ne the same
2. k1, k2, k3 must not depend on i for g1, g2, g3 to genarate arithmetic progressions, respectively. Also, g1, g2, g3 must have the forms indicated so that the values generated by all three form one arithmetic progression.√√ The next theorem deals with the opposite direction namely the replacement of concatenators by imbricators.

THEOREM 3.8:

A concatenator of the form

(Conc) #i:1,n,1[p(i) @]

may always be replaced by the imbricator

(imbr) #i:1,n,1[p(i) @ @]

Proof:

The expansion of both replicators (Conc) and (Imbr) is given by

$$\overset{n}{\underset{i=1}{COPY}}\{p(i)/\ /\}$$

Also by lemma 7, by replacing a generalized element by a macro sequence, and afortiori by another generalized element a macro sequence is obtained. One more thing has to be proved: that "p(i) @ @" may be produced by "imbrseq".

The string "p(i) @" may either be produced by

1. {morelement ;}* concor or by
2. {morelement ;}+ @

The string "p(i) @" may therefore be of the forms

1'. {morelement ;}* gelement,...,@
2'. {morelement ;}+ @

If an "@" is appended to 1' and 2' ,i.e. constructing "p(i) @ @", the new strings may be produced by applying the production options 11 or 12 of "imbr_at_seq". √√√

Some important corollaries of the above theorem are the following:

COROLLARY:

At the expense of one extra symbol namely "@", all concatenators may be replaced by imbricators.

COROLLARY:

Concatenators can only generate sequences which can be generated by imbricators also.

An important implication would be that the syntax rules for sequence

replicators could be simplified by eliminating production rules for concatenators without reducing the generality of the notation. However, this may only be done at the expense of some loss of conciseness, since an extra "@" has to be used.

We may also constrain the syntax for imbricators to produce just genuine imbricators for which the conditions of theorem 3.8 do not apply. This could be done by forcing at least one pair of parentheses in "[ ]" to open before the first "@" and close after the second "@". Non-genuine imbricators are produced when the whole string produced by "imbrseq" is produced by "imbr_at_seq". We decided against that for it would worsen the conciseness and readability of programs. Consider for example the n-free frame buffer

NP12 #i:1,n,1[path DEPOSIT(i);REMOVE(i) end]

which could be modified to a fill-empty last in first out queue by adding the path

NP20 path #i:1,n,1[DEPOSIT(i);@;@;REMOVE(i)] end

which involves a non-genuine imbricator which could be replaced by two concatenators as follows:

NP21 path #i :1,n,1[DEPOSIT(i);@] ; #i :n,1,-1[REMOVE(i);@] end

The path NP21 with the two concatenators is the least concise. The replacement of an imbricator by two concatenators may not be as simple as the above example. It could lead to long index specifications which are difficult to read as the following example demonstrates. Let us consider the bodyreplicator

NP22 #i:1,10,1[path #j:1,100,i[A(j);@;@;B(j)] end

which generates ten paths with their sequences consisting of just a non-genuine imbricator. If we replace the imbricator by the two concatenators

```
#j1:1,100,i[A(j1);?] ; #j2:100,1,-i[B(j2);?]
```

a different sequence will be generated. We can easily see that when i=2. Then the range of index j1 is

```
1,3,...,97,99
```

but that of j2 is

```
100,98,...,4,2
```

and not

```
99,97,...,3,1
```

as its correct range should be. The reason for this difference is that the index specification of j2 should be correctly specified by

```
#j2:((100-1)//i)*i+1,1,-i
```

which is quite a complicated formula. The above index specification gives the same range as the erroneous one when

```
((100-1)//i)*i+1=100     =>
(100-1)//i=(100-1)/i     =>
99//i=99/i
```

that is when

```
99 mod i=0
```

for values of i between 1 and 10, that is for i=1,3,9.

3.3.2 The Expansion of Distributors

In section 3.2.7 we developed the syntax for distributors, we defined compatibility criteria (CC1) and (Drest1) for well defined distributors,

and we sketched their expansion. In this section we formally define their expansion and we prove that for each distributor there are sequence replicators which yield the same string after expansion and we derive necessary and sufficient conditions for obtaining sequence replicators from distributors yielding the same string after expansion.

Let us first, obtain the rule for the expansion of distributors without specifying any subrange, i.e. distributors of the form

ND14 sep[p]

where p is a macro sequence of array slices (cf. section 3.2.7). For the expansion of a distributor in this form to be defined, the first compatibility criterion (CC1) must be obeyed, implying that all the distributable dimensions of array slices must contain the same number of sections, say m. In the expansion of ND14 m copies of p will be generated separated by "sep" which may be formally obtained by

(E1) $\overset{m}{\underset{j=1}{\text{COPY}}}\{p\ sep/\ /\}$

Furthermore, the array slices of the first copy must be replaced by the first array section of this slice, in the second copy by the second section etc. Therefore, the blank fields of the array slices of the p's in (E1) must be replaced by a function of j which relates the j'th copy with the j'th section of each array slice. We specify that by

(E2) $\overset{m}{\underset{j=1}{\text{COPY}}}\{p(g(j))\ sep/\ /\}$

in which p(g(j)) indicates that each of the distributable dimensions of ND14 must be replaced by a function g(j).

The function g should be such that for a particular slice

g(j) should give the index of the j'th section for j=1,2,...,m

The function g for each particular slice may only be obtained from the collectivisors since these define the sections and their order. The

collectivisors may be of two forms: those produced by the non-terminal "simpleardecl" and those produced by "replardecl". All collective names declared by the former are of the form:

```
X(lb1:ub1,...,lbn:ubn)
```

where "lbi" and "ubi" for i=1,...,n denote the lower bound and upper bound respectively of dimension i of collective name X, may be declared equivalently by the latter as follows:

```
#j1:lb1,ub1,1[...#jn:lbn,ubn,1[X(j1,...,jn)]...]
```

or when all replicators are transformed into their normal form by

```
#j1':1,m1,1[...#jn':1,mn,1[X(lb1+j1'-1,...,lbn+jn'-1)]...]
```

where mi=ubi-lbi+1 for i=1,2,...,n. From now on we assume that all collective names are declared by collectivisors produced by "replardecl" in which all replicators are in their normal form. For example the collective names A and B declared by NC7 may be equivalently declared by

```
NC11 array #k:1,4,1[A(k-1)]
            #k:1,4,1[#j:1,3,1[B(k,j)]]
     endarray
```

or more concisely by

```
NC12 array #k:1,4,1[A(k-1) #j:1,3,1[B(k,j)]]endarray
```

In general the declaration of subscripted operations corresponding to a collective name Y of n dimensions has the form

```
(NCR) #k1:1,m1,1[...#kn:1,mn,1[Y(h1(k1),...,hn(kn))]...]
```

where hi for i=1,2,...,n are integer functions of ki.

The order of the sections of an array slice along its i'th dimension for i=1,...,n may be easily obtained by:

hi(j) gives the index of the j'th section for j=1,...,mi
along dimension i, for i=1,...n.

This implies that hi(j) is the function by which g(j) in each of the operations of p in (E2) must be replaced.

Let now us expand ND1

ND1 ;[A,B(,3)]

where A, B have been declared by NC12 using the form (E2). To construct "p(g(j))" from p which in this case is "A,B(,3)" we have to replace A by A(j-1) and B(,3) by B(j,3) where the expressions j-1 and j have been obtained from NC12. The expansion of ND1 is given by

(E3) $\overset{4}{\underset{j=1}{\text{COPY}}}${A(j-1),B(j,3);/ /}

which yields

A(0),B(1,3);A(1),B(2,3);A(2),B(3,3);A(3),B(4,3)

Let us also expand the distributor

ND15 ;[C]

where C is declared by NC13

NC13 array #j:1,4,1[C((j-1)**2)] endarray

where "**" denotes "to the power", by which the operations

C(0), C(1), C(4), C(9)

are declared. The expansion of ND15 is given by

```
 4
COPY{C((j-1)**2);/ /}
j=1
```

which yields

```
C(0);C(1);C(4);C(9)
```

Let us now obtain a formula for the expansion of a distributor with subrange

```
ND2 sep#inind,fiind,incind[p]
```

According to the compatibility criterion (Drest1) of distributors, distributors are well defined when all the distributable dimensions of array slices on which the distributor applies contain at least Ns sections, where Ns=(fiind-inind)//incind+1. The expansion of ND2 will be given by

```
      Ns
(E5) COPY{p(g(1)) sep/ /}
     l=1
```

in which p(g(1)) indicates that each of the blank fields of distributable dimensions on which the distributor operates have been replaced by a function g(1). This function will not in general be the same as in ND1 since

g(1) must give the inind'th array slice,
g(2) the (inind+incind)'th slice,
etc.

The order of the sections we would like to generate may be given by the formula

```
f(l)=inind+(l-1)*incind, for l=1,2,...,Ns =(fiind-(inind))//(incind)
```

Therefore the function g(1) for dimension i, $1 \leq i \leq n$ in (E5) will be

```
g(l)=hi(f(l))
```

where hi is obtained from the declaration of the form of (NCR).

Let us demonstrate this rule by expanding

```
ND16 ;#2,4,2[D]
```

where D is defined by NC14

```
NC14 array #i:2,11,2[D(i)] endarray
```

which when the replicator is transformed into its normal form (cf. 3.3.1), is declared by

```
array #i1:1,5,1[D(2+(i1-1)*2)] endarray
```

or more simply by NC15

```
NC15 array #i1:1,5,1[D(2*i1)] endarray
```

defining operations

```
D(2), D(4), D(6), D(8), D(10)
```

The expansion of ND16 is given by:

$$\overset{Ns}{\underset{1=1}{COPY}}\{D(2+(f(1)-1)*2);/\ /\}$$

where Ns=(4-2)//2+1=2 and f(1)=2+(1-1)*2. Therefore (E6) becomes

$$\text{(E7)}\quad \overset{2}{\underset{1=1}{COPY}}\{D(2+(2+(1-1)*2-1)*2);/\ /\}=$$

$$\overset{2}{\underset{1=1}{COPY}}\{D(2+(2*1-1)*2);/\ /\}=$$

$$\overset{2}{\underset{1=1}{COPY}}\{D(4*1);/\ /\}=D(4);D(8)$$

Finally, let us expand ND17

```
ND17 ;#2,4,2[C]
```

where C is declared by NC13. Its expansion is given by (E8)

```
(E8)  Ns
      COPY{C((f(1)-1)**2);/ /}
      l=1
```

where Ns=(4-2)//2+1=2 and f(1)=2+(1-1)*2 which when substituted in (E8) we obtain (E9)

```
(E9)  2
      COPY{C((2+(1-1)*2-1)**2);/ /}=
      l=1

       2
      COPY{C((2*1-1)**2);/ /}=
      l=1

      C(1);C(4)
```

A distributor of the form

```
ND1 sep[p]
```

may be considered as a special case of ND2

```
ND2 sep#inind,fiind,incind[p]
```

in which inind=1, incind=1 and fiind is the common number of sections in the distributable dimensions in the array slices of p.

Then the expansion distrexp^0(D) of a distributor D of the form of ND2 in which inind, fiind, incind may be declared implicitly, is given by

```
 Ns
COPY{p(g(j)) sep/ /}
j=1
```

where Ns=(fiind-inind)//incind+1, p(g(j)) indicates that each blank field of distributable dimensions on which the distributor operates, will be replaced by a function

```
g(j)=hi(f(j))
```

where f(j) is defined by

f(j)=inind+(j-1)*incind

and hi is obtained from the corresponding i'th dimension of each collective name in the collectivisors.

For the expansion of distributors to be non empty the following restriction (Drest3) must hold:

(Drest3)

incind≠0 and Ns=(fiind-inind)//incind+1≥1

that is at least one copy of the regularity must be made. For the expansion of a distributor to generate subscripted operations which are permitted by the collectivisors the following restriction (Drest4) must be imposed on the values of "inind", "fiind", "incind".

(Drest4)

1≤inind+(j-1)*incind≤Ms for j=1,...,Ns

where Ms is the minimum number of slices over the number of slices of all the distributable dimensions of the distributor. The expression

f(j)=inind+(j-1)*incind

gives the array slice of ordinality j, 1≤j≤Ns and as such must take values between 1 and Ms. Let us obtain restrictions on the values of "inind", "fiind" and "incind" independent of j. As f(j) is a monotonic function its lower and upper values are obtained for j=1 and j=Ns. As f(j) may be either increasing or decreasing with increasing values of j we may only infer that for j=1

1≤inind+(1-1)*incind≤Ms ⇒
1≤inind≤Ms (I)

and that for j=Ns

$1 \leq inind+((fiind-inind)//incind)*incind \leq Ms \quad \Rightarrow$

$1 \leq inind+((fiind-inind)/incind-e)*incind \leq Ms \quad (-1<e<1) \Rightarrow$

$1 \leq inind+fiind-inind-e*incind \leq Ms \quad \Rightarrow$

$1 \leq fiind-e*incind \leq Ms \quad \Rightarrow$

$1+e*incind \leq fiind \leq Ms+e*incind$

When incind>0, the expression 1+e*incind is minimum when e tends to -1, and the expression Ms+e*incind is maximum when e tends to 1. Therefore,

when incind>0 : $1-incind<fiind<Ms+incind$ (IIa)

When incind<0 the the expression 1+e*incind is minimum when e tends to 1 and the the expression Ms+e*incind is maximum when e tends to -1. Therefore

when incind<0 : $1+incind<fiind<Ms-incind$ (IIb)

By combining (IIa) and (IIb) we obtain:

$1-|incind|<fiind<Ms+|incind|$ (II)

A distributor may be considered as a shorthand for some replicators. In fact for every distributor there is a family of replicators which when expanded generate the same string as the string obtained from the expansion of the distributor. Before we formally show how to obtain such replicators let us prove the following lemma.

LEMMA 8:

If s(i) is a macro sequence involving integer expressions depending on some integer i then the concatenator

#i:1,n,1[s(i) sep @]

is syntactically well-formed.

Proof:

Since s(i) is a macro sequence it may in general be of the form

morelement1 ; ... ; morelementm

If sep is a semicolon the string s(i) sep @ may be produced by the second option of "concseq"and if sep is a comma by the first option of "concseq". ///

Let us now prove the theorem for the replacement of distributors by concatenators.

THEOREM 3.9:

The string obtained by the expansion of a distributor of the general form

ND2 sep#inind,fiind,incind[p]

may also be generated by a concatenator of the normal form

#j:1,Ns,1[p(g(j)) sep @]

where Ns=(fiind-inind)//incind+1 and p(g(j)) is obtained from p by substituting the fields of the distributable dimensions of sections in ND2 by g(j) as in distrexp^0.

Proof:

The syntax rules in 3.2.7 for distributors specify that the string p is produced by "msequence" in which the fields of the distributable dimensions on which the distributor operates are blank. When p is transformed to p(g(j)) all these blank fields are replaced by an integer expression g(j). Any other blank fields in p(g(j)) are in array slices which correspond to other distributors nested in ND2.

By lemma 8 and lemma 1 the replicator of the form

```
#j:1,Ns,1[p(g(j)) sep @]
```

is syntactically well-formed since p(g(j)) is a macro sequence. Furthermore its expansion is given by

$$\overset{Ns}{\underset{j=1}{COPY}}\{p(g(j))\ sep/\ /\}$$

which is the same as the formula for the expansion of ND2. If the values of "inind", "fiind" and "incind" satisfy (Drest3) and (Drest4) then the concatenator generates only indexed operations permitted by the collectivisors.√√√

The concatenator constructed in the above theorem is in normal form and from it all concatenators belonging to its family may be obtained, all expanding to the same string. This concatenator may by theorem 3.8 be replaced by an imbricator generating the same strings. The concatenators generating the same strings as ND1, ND16, and ND17 are

```
NR12 #j:1,4,1[A(j-1),B(j,3);@]
NR13 #1:1,2,1[D(4*1);@]
NR14 #1:1,2,1[C((2*1-1)**2);@]
```

respectively.

A corollary of theorems 3.3 and 3.9 is

COROLLARY:

All distributors expand to macro sequences.

The reverse of theorem 3.9 that all replicators expand to strings which may be generated by distributors does not hold. However, under some conditions this is possible. The next theorem for the replacement of concatenators and imbricators by distributors gives these conditions.

THEOREM 3.10:

The string generated by the expansion of a sequence replicator of the form

(NR)#i:1,n,1[p(i) @ t @ q(i)]

may be generated by one distributor if and only if the following conditions hold:

1. either t and one of p or q are empty or the conditions of theorem 3.7′ hold.

2. $n \geq 1$

3. all operations in p and q must be subscripted by i and the index "i" should not be involved in expressions in replicator index specification or subrange specifications of replicators and distributors in p or q.

4. each subscripted operation must have exactly one field depending on the index i by an expression say g(i).

5. if p(i) and q(i) involve distributors, the field of subscripted operations depending on index "i" should be in the same position relative to the blank fields of the array slices of these distributors.

6. the g(i) in each of the above fields must be such that it may be transformed into the form h(f(i)) where h is the function in the corresponding dimension of each collective name in the collectivisors and f(i) is the same for all fields and is of the form a+(i-1)*b where a, b are integers with the restriction that $b \neq 0$.

Proof:

(if)

If t and one of p or q in (NR) are empty and since $n \geq 1$, the expansion of (NR) is nonempty and according to theorem 3.7 the expansion of (NR) may be obtained by a single concatenator. Similarly, if p, t, q satisfy the conditions of theorem 3.7′ then the expansion of (NR) is nonempty and may be also be obtained by a single concatenator.

either #i:1,n,1[p(i) @]
or #i:n,1,-1[q^1(i) @]

where q^1(i) is defined in theorem 3.7. Therefore it suffices to show that the expansion of a concatenator of the form

(NR1) #j:1,m,1[s(g(j)) sep @]

where s(g(j)) indicates that the sixth condition holds, may be obtained by a distributor.

Let us construct the distributor (ND)

(ND) sep#a,a+(m-1)*b,b[s]

where s is obtained from s(g(j)) by removing from each subscripted operation the integer expression g(j). Since s(g(j)) is a macro sequence in which all operations are subscripted by condition 3, by lemma 2, s is a macro sequence having array slices instead of operations. The index i does not appear anywhere is s as it has been eliminated from the subscripted operations and as any other context it could be in, has been excluded by condition 2. By condition 3, the distributor ND applies to a single distributable dimension of each slice in s.

If "sep" in ND does not apply to the right field of the slices of s then we may use the section selection feature of the distributor either in ND or in the distributors in p(i) and/or q(i). We know this is possible because of condition 5.

Obviously its expansion is given by

$$\overset{m}{\underset{j=1}{\text{COPY}}}\{s(g(j))\ \text{sep}/\ /\}$$

since the number of times s will be copied is given by

(a+(m-1)*b-a)//b+1=
((m-1)*b)//b+1=m

and g(j) is the same as h(f(j)). The same formula gives also the expansion of (NR1).

(only if)

Condition 1 must be satisfied for
otherwise more than one of the arguments of the COPY expression giving its expansion will be non empty. The COPY expression giving the expansion of a distributor has the second and third arguments empty.

Condition 2 guarantees that the expansion of (NR) is non empty which is necessary since distributors should not generate empty expansions.

All operations should be subscripted by expressons depending on "i", as condition 3 requires, for after these expressions are eliminated all operations inside "[ ]" of ND should be array slices. Also "i" should not be involved anywhere else, for the index "i" inside "[ ]" of ND, it will still be by context an integer constant, not controlled at all by the distributor and therefore undefined.

Subscripted operations in s must have at most one field depending on i, as condition 4 requires, as each distributor applies only to one dimension of each array slice.

Condition 5 must hold since otherwise, neither the default rule nor the section selection feature of distributors can specify the right slices to be distributed.

Finally, g(i) must be of the specified form since the subrange of distributors selects sections of array slices the position ordering of which form arithmetic progressions.√√√

Let us demonstrate how distributors may be obtained which generate the same strings as replicators. Consider the concatenator NR12

```
NR12 #j:1,4,1[A(j-1),B(j,3);@]
```

where collective names A and B have been declared by NC12. The

replicator NR12 satisfies the first five conditions of theorem 3.10. Let us demonstrate that it also satisfies the sixth condition. The expression (j-1) subscripting A should be possible to be re-writen as ha(g(j)) where f(j) is of the form a+(j-1)*b and ha is the function subscripting A in NC12. Therefore

ha(f(j))=f(j)-1

As f(j) is of the form

f(j)=a+(j-1)*b

we have to find integers a and b such that

a+(j-1)*b=j or
a-b+b*j=j

The only such integers are a=b=1. Therefore the expression j-1 may be re-writen as

f(j)-1 or as
1+(j-1)*1-1

Therefore condition 6 is satisfied by A(j-1). It should also be satisfied by B(j,3). Simirarly, the expression j should be re-writen as hb(f(j)) where hb is subscripting the first dimension of B in the collectivisor NC12. As

hb(f(j))=f(j)

we have to verify that

f(j)=1+(j-1)*1

is the same as

f(j)=j

which obviously is. Therefore condition 6 is also satisfied and the distributor expanding to the same string as NR12 is ND18

```
ND18 ;#1,4,1[A,B(,3)]
```

or more simply

```
ND1 ;[A,B(,3)]
```

as the subrange of ND18 is redundant defining all sections of array slices of A and B(,3).

Let us also examine the concatenator NR13

```
NR13 #1:1,2,1[D(4*1);@]
```

where D is declared by NC14. The replicator NR13 satisfies the first five conditions of thoerem 3.10. Let us try to transform "4*1" into the necessary form h(f(1)). Since f(1) must be of the form a+(1-1)*b the relation

```
g(1)=4*1
```

must hold. Since h from the collectivisor NC14 is h(j)=2+(j-1)*2 the relation

```
2+(f(1)-1)*2=4*1
```

must hold which implies that

```
f(1)=(4*1-2)/2+1=2*1
```

Since f(1) must be of the form

```
f(1)=a+(1-1)*b
```

the relation

```
a+(1-1)*b=2*1
```

must hold, which implies that

```
a=b=2
```

Therefore f(1)=2+(j-1)*2 and NR13 satisfies the sixth condition of theorem 3.10. The distributor expanding to the same string as NR13 is ND16

```
ND16 ;#2,4,2[D]
```

Not all g(i) may be tranformed into the appropriate form. Consider for example the replicator

```
NR15 #i:1,3,1[E(i**2);@]
```

where E is declared by NC16

```
NC16 array #j:1,10,1[E(j)] endarray
```

The concatenator NR11 satisfies the first five conditions. Let us try to transform g(i)=i**2 into the appropriate form, h(f(j)). As h(j)=j

```
h(f(i))=f(i)
```

Therefore f(i) must be the sane as i**2 and f(i) must be of the form

```
f(i)=a+(i-1)*b
```

which means that

```
a+(i-1)*b=i**2
```

must hold. But there are no integers a and b for which this relation holds, as the left hand side is a linear expression of i whilst the right hand side a quadratic expression of i. Therefore condition 6 is

not satisfied and there is no distributor which may generate the string which NR13 generates. The reason is that NR15 generates the string

E(1);E(4);E(9)

that is, consisting of the first, fourth and ninth operations of E the ordering of which does not form arithmetic progression.

3.3.3 The Expansion of Macro Programs

In the previous two sections 3.3.1 and 3.3.2 we obtained expansion rules for replicators (replexp^0) and distributors (distrexp^0) and we proved various properties which their expansions possess. Here we define the complete expansion of macro programs and by using the results theorems 3.3., 3.4, 3.5 and the corollary of theorem 3.9 of the previous two sections we show that their expansion yields basic COSY programs.

Let us represent a macro program schematically using syntactic variables to represent its syntactic entities, that is substrings produced by non-terminals. A macro program will be denoted by MPROG and represented by

program MPBODY endprogram

where MPBODY denotes a substring produced by the non-terminal "mprogrambody". As such MPBODY may have the form

CPQBR1 ... CPQBRn

where each CPQBRi for i=1,...,n denotes a single path or process or bodyreplicator possibly headed by collectivisors. If headed by collectivisors it may be represented by

COLs PQBR

where COLs denotes a collection of collectivisors and PQBR a single path, or a process or a bodyreplicator. A bodyreplicator may be

represented by

#i:1,n,1[PQBRs]

where PQBRs denotes a collection of paths, processes and bodyreplicators. A bodyreplicator upon expansion generates a collection of paths, processes and bodyreplicators represented by

PQBR1 ... PQBRn

where each PQBRi for i=1,...,n denotes a single path or process or bodyreplicator. A path and a process will be represented by

path MSEQ end
process MSEQ end

respectively, where MSEQ denotes a macro sequence. A macro sequence will be represented by

MOR1 ;...; MORn

where each MORi for i=1,...,n denotes a macro orelement, which is represented by

GEL1 ,..., GELn

where each GELi for i=1,...,n denotes a generalized element.

In general, a generalized element may involve right and left replicators and will be represented by

RRs M LRs

where RRs and LRs denote right and left replicators respectively and M denotes either a starelement or a sequence replicator or a distributor. A generalized element may be just a sequence replicator denoted by SREPL, or a distributor denoted by DISTR, or a starelement represented by

EL* or EL

where EL denotes an element which could be either an operation or an indexed operation denoted by OP or a macro sequence in parentheses represented by

(MSEQ)

The complete expansion of a macro program MPROG is given by expand(MPROG) where the function "expand" is defined as follows:

expand(e)=cases e:

1.	program MPBODY endprogram	→ program expand(MPBODY) endprogram
2.	CPQBR1 ... CPQBRn	→ expand(CPQBR1) ... expand(CPQBRn)
3.	COLs PQBR	→ expand(PQBR)
4.	#i:1,n,1[PQBRs]	→ expand(replexp^0(#i:1,n,1[PQBRs]))
5.	PQBR1 ... PQBRn	→ expand(PQBR1) ... expand(PQBRn)
6.	path MSEQ end	→ path expand(MSEQ) end
7.	process MSEQ end	→ process expand(MSEQ) end
8.	MOR1;...;MORn	→ expand(MOR1);...;expand(MORn)
9.	GEL1,...,GELn	→ expand(GEL1),...,expand(GELn)
10.	RRs M LRs	→ expand(gelexp^0(RRs M LRs))
11.	SREPL	→ expand(replexp^0(SREPL))
12.	DISTR	→ expand(distrexp^0(DISTR))
13.	EL*	→ expand(EL)*
14.	OP	→ OP + possible expression evaluations
15.	(MSEQ)	→ (expand(MSEQ))

We may now prove the theorem for the expansion of macro programs to basic programs.

THEOREM 3.11:

The expansion of a macro program MPROG produced by the syntax rules of section 3.2 given by

expand(MPROG)

is a well-formed basic COSY program.

Proof:

The expansion certainly stops.

We shall prove the theorem for each of the fifteen cases of syntactic entities on which function "expand" applies.

case 1

Applying expand to MPROG we obtain

expand(program MPBODY endprogram)=program expand(MPBODY) endprogram

which is a basic program since expand(MPBODY) is a basic programbody as may be shown by case 2.

case 2

Applying expand to MPBODY we obtain

expand(CPQBR1 ... CPQBRn)= expand(CPQBR1) ... expand(CPQBRn)

The r.h.s. is a basic programbody since each of

expand(CPQBRi) for i=1,...,n

is a basic programbody as may be shown by case 3.

case 3

Applying expand to a single path or process or bodyreplicator headed by a collectivisor we obtain

expand(COLs PQBR)=expand(PQBR)

which is a basic programbody as may be shown by cases 4, 6, 7.

case 4

Applying expand to a bodyreplicator we obtain

expand(#i:1,n,1[PQBRs])=expand(replexp^0(#i:1,n,1[PQBRs]))

Since

replexp^0(#i:1,n,1[PQBRs])

yields a collection of paths, processes and bodyreplicators in which the index "i" has been replaced by values in the range of "i", its expansion may be shown by case 5 to be a basic programbody.

case 5

Applying expand to a collection of paths, processes and bodyreplicators we obtain

expand(PQBR1 ... PQBRn)=expand(PQBR1) ... expand(PQBRn)

which is a basic programbody since each of

expand(PQBRi) for i=1,...,n

is a basic programbody as may be shown by cases 4, 6, 7.

case 6

Applying expand to a macro path we obtain

expand(path MSEQ end)=path expand(MSEQ) end

which is a basic path and a basic programbody if

expand(MSEQ)

is a basic sequence which may be shown by case 8.

case 7

Similarly, the expansion of a macro process is a basic process and a basic programbody.

case 8

Applying expand to a macro sequence we obtain

expand(MOR1;...;MORn)=expand(MOR1);...;expand(MORn)

Each of

expand(MORi) for i=1,...,n

is a basic sequence as may be shown by case 9.

Therefore, the r.h.s. is a basic sequence since if s1 and s2 are basic sequences "s1;s2" is a basic sequence also. This may be shown by similar arguments to that of lemma 3.

case 9

Applying expand to a macro orelement we obtain

expand(GEL1,...,GELn)=expand(GEL1),...,expand(GELn)

Each of

expand(GELi) for i=1,...,n

is a basic sequence as may be shown by each of the following cases.

Therefore, the r.h.s. is a basic sequence also since if s1 and s2 are basic sequences "s1,s2" is a basic sequence also. Again this may be shown by similar arguments to those of lemma 3.

case 10

Applying expand to a generalized element which involves left and/or right replicators we obtain

expand(RRs M LRs)=expand(gelexp^0(RRs M LRs))

As we have shown in theorem 3.5 the expansion of a generalized element

gelexp^0(RRs M LRs)

is a macro sequence in which all the indices of the left and right replicators have been replaced by integer values in their range. The expansion of this macro sequence may be shown to be a basic sequence by case 8.

case 11

Applying expand to a sequence replicator we obtain

expand(SREPL)=expand(replexp^0(SREPL))

By theorems 3.3 and 3.4 the expansion of a sequence replicator given by

replexp^0(SREPL)

yields a macro sequence in which the index "i" has been replaced by integer values in its range. By case 8, its expansion may be shown to be a basic sequence.

case 12

Applying expand to a distributor we obtain

expand(DISTR)=expand(distrexp^0(DISTR))

By the corollary of theorem 3.9 the expansion of a distributor given by

distrexp^{0}(DISTR)

yields a macro sequence in which all the distributable dimensions of the distributor have been replaced by integer values. The expansion of this macro sequence may be shown to be a basic sequence by case 8.

case 13

Applying expand to a starelement we obtain

expand(EL*)=expand(EL)*

which is a basic element since expand(EL) is a basic element as may be shown by cases 14, 15.

case 14

Applying expand to a simple or a subscripted operation we obtain

expand(OP)=OP + possible expression evaluations

which is a basic operation.

case 15

Applying expand to an element of the form (MSEQ) we obtain

expand((MSEQ))=(expand(MSEQ))

which is a basic element since

expand(MSEQ)

is a basic sequence as may be shown by case 8.

As we have considered every possible case of syntactic entities of macro macro programs to which the function "expand" applies, we may conclude that the theorem is proven.√√√

In this section 3.3 we formally defined the expansion of replicators, distributors and of complete macro programs. We prove that concatenators, imbricators and distributors generate macro sequences upon their expansion. We also proved that the expansion of complete macro programs yields well-formed basic programs. We also proved a number of theorems for the replacement of macro elements in macro sequences by other macro elements.

3.4 EVALUATION OF THE NEW NOTATION FOR MACRO COSY

In the previous two sections 3.2 and 3.3 we introduced a new macro notation and grammar, we defined and characterized the expansion of replicators, distributors and macro programs. In this section we evaluate this new notation using as criteria the four properties we set at the begining of section 3.2 which a "good" macro notation should possess.

1. As we have proved in section 3.3.3 programs produced by the grammar of section 3.2 always generate well-formed basic programs when expanded. This grammar gives context-free rules and no meta-restriction rules are required to constrain the regularities of replicators. The few meta-restrictions imposed are of a context-sensitive nature and cannot be expressed by context-free rules. These include the restrictions that collective names should be declared before any of its corresponding subscripted operations are used in paths or processes; that the number of dimensions of indexed operations corresponding to a collective name should have the same number of dimensions as specified in the collectivisors, etc. The production of macro programs which always yield well-formed basic programs when expanded was considered to be a very important property of a macro notation.

However, this property on its own does not justify a good macro notation as the macro features it involves should generate a large class of strings in order to represent basic COSY strings concisely.

2. We shall examine in detail the generality of each feature of the notation, the generality of the collectivisor, the bodyreplicator, the sequence replicators, the left and right replicators and the distributor.

a. the collectivisor

Collectivisors do not generate any basic strings as they are eliminated upon the expansion of macro programs, but declare permissible sets of subscripted operations. They are important though, since the expansion of distributors depends on these declarations; distributors do not explicitly generate indices but generate the indices defined by the collectivisors.
They are also usefull as a check for the indices used in the rest of programs.

Collectivisors may declare rectangular arrays of any number of dimensions either specifying the lower bound in each of these explicitly or assuming it to have the value one implicitly. By using replicators generating permissible sets of subscripted operations other shapes of arrays may be declared. Although, more complex shapes could be permitted to be declared we did not allow the maximum degree of generality possible and we imposed the restrictions (Crest3) specifying that there should be as many dimensions in an indexed operation in collectivisors as the number of replicators within it is nested and that indices in each dimension should depend directly on one distinct replicator index. These restrictions were imposed to guarantee the independence of the indices and to avoid duplication of declaration of subscripted operations. A third and more subtle reason for these restrictions was to avoid the declaration of collective names, one dimension of which either depends directly on two or more indices, or depends on one index which itself depends on another index on which none of the indices in other dimensions depend directly. These collectivisors would overcomplicate the expansion of distributors which would no longer be replaced by a single concatenator but by a number of them nested within each other. Consider for example the declarations

```
C array
    #i:0,9,1[#j:0,9,1[A(100*i+j)]]
    #i:0,9,1[#k:100*i,100*i+9,1[B(k)]]
  endarray
```

which are not permitted by our restrictions. These declare two one-dimensional arrays A and B the indices of which take the values:

0,1,...,9, 100,101,...,109, ... , 900,901,...,909

The only difference in the declarations of A and B is that the indices of A depend on i and j directly and the indices of B directly on k and indirectly on i. With the above declarations of A and B the distributor

```
D ;[A,B]
```

would no longer be replaced by a single concatenator but by two nested ones as follows:

```
#i:0,9,1[#j:0,9,1[A(100*i+j),B(100*i+j);@];@]      or
#i:0,9,1[#k:100*i,100*i+9,1[A(k),B(k);@];@]
```

Although, this kind of declarations increase the class of strings distributors could generate we have excluded them for they would overcomplicate the expansion rules for distributors. The above arrays A and B could be modified to the two dimensional arrays A1 and B1 declared by:

```
NC17 array #i:0,9,1[#k:100*i,100*i+9,1[A1(i,k) B1(i,k)]] endarray
```

The above collectivisor is valid in the new notation, as the number of dimensions of A1 and B1 are the same as the number of replicators defining them and each dimension depends on a single replicator index directly. The operations in A and B correspond to operations in A1 and B1 as follows:

```
A1(i,k) and B1(i,k) correspond to A(k) and B(k) respectively,
                    for i,k as generated by the replicators in NC17
```

Under the restrictions Crest3 on collectivisors, the expansion of distributors is reasonably simple which compensates for the loss of generality of the strings distributors may generate. Under the above correspondence of operations of A and B with operations of A1 and B1 the string the distributor

D ;[A,B]

generates, when A and B have been declared by C, may be generated by by the distributor ND19

ND19 ;[;[A1,B1]]

where A1 and B1 are declared by NC17. The distributor ND19 is permitted in the new macro notation.

Collectivisors contribute to the notation the option of simplifying macro sequences by using distributors rather than the more lengthy replicators. Collectivisors in a program without distributors are not essential and they may only serve as a means of testing that replicators do not generate subscripted operations not admitted by collectivisors.

b. the bodyreplicator

Bodyreplicators may generate paths and processes and other bodyreplicators. Unlike sequence replicators they are only of one form, generating consecutive regularities. In that respect they are analogous to concatenators and not to imbricators. We did not allow two types of bodyreplicators "bodyconcatenators" and "bodyimbricators" for two reasons. The first is a pragmatic one; we have never needed or used bodyimbricators although some of the grammars permitted them [TL77, LT76]. The second is that the paths and/or processes a "bodyimbricator" generates may be generated by a single "bodyconcatenator", as we have indicated in section 3.1.3 where discussing the syntax for bodyreplicators of [LT76].

c. the sequence replicators

They may generate regularities which are macro sequences. We have distinguished two kinds of these replicators, concatenators and imbricators. In the expansion of concatenators all regularities follow each other and these only differ in the subscripts of the indexed operations they involve. In the expansion of imbricators regularities wrap or imbricate each other. All but one of these regularities differ in the integer expressions they involve. The innermost regularity however differs additionally from the rest in that instead of imbricating another regularity it imbricates a string, namely that between the two "@"s in an imbricator, dropping the separators before and after the two "@"s. This imbricator is a powerful extension we have introduced and allows generation of sequences which cannot be generated by a single replicator produced by any other grammar.

We have excluded replicators which do not generate matching pairs of parentheses. These are the range, context and neighbourhood dependent replicators. Of the three only the third has been used in macro programs but for a very specific purpose: to specify the more general imbrication of regularities which our imbricators do permit. For example the stack with a test for "full" had to be specified by using two wide concatenators, which are neighbourhood dependent, as follows:

P74 path #i:1,n,1[(UP(i);@] ; full* ; #i:n,1,-1[DOWN(i))*;@] end

The string obtained from the expansion of the two neighbourhood dependent replicators may be generated by one of the new imbricators:

NP23 path #i:1,n,1[(UP(i);@;full*;@;DOWN(i))*] end

Since no other use was made of these replicators we have not obtained formal results on the limitation of our notation due to their elimination. We however outline how macro sequences involving these replicators may be transformed into macro sequences valid in our notation. Although, these replicators do not generate sequences their expansion together with their context should form sequences, parts of which may be generated by sequence replicators.

Simple range dependent replicators where "(" or ")" is the immediate left and respectively right context of them may be generated together with their context by a single imbricator. For example, the string

(((#i:1,3,1[A(i);D(i)),@]

after the expansion of the replicator becomes the sequence

(((A(1);D(1)),A(2);D(2)),A(3);D(3))

which may be generated by the imbricator

NR16 #i:3,1,-1[(@@,A(i);D(i))]

The context of the range dependent replicators could be more involved in that "(" and ")" are not their immediate context, as for example:

(a;(b;(c;#i:1,3,1[D(i)),@]

which after the expansion of the replicator becomes

(a;(b;(c;D(1)),D(2)),D(3))

The above sequence cannot be generated by any of our replicators since the regularities differ in the simple operation names and not just in the subscripts of indexed operations. We may however use a collective name, say A, corresponding to the subscripted operations A(1), A(2), A(3) and rename the simple operations a, b, c to A(3), A(2) and A(1) respectively. Thus the above string becomes

(A(3);(A(2);(A(1);D(1)),D(2)),D(3))

which may be generated by the imbricator

NR17 #i:3,1,-1[(A(i);@;@,D(i))]

Not all well-formed basic strings parts of which are generated by range

dependent operations may be generated by sequence replicators. Consider for example the string

(a;(b,(c;#i:1,3,1[D(i));@]

in which the connectives after a and c are ";" but the connective after b is ",". Here the mapping of operations to indexed operations is not sufficient to overcome the problem of constructing a sequence replicator expanding to the string above.

A string s part of which is generated by a range dependent replicator which is a wide concatenator may be generated by an imbricator provided that the head of the string s not generated by the range dependent replicator may be generated by a range dependent replicator. Then if the ranges of the indices of the two wide concatenators are the same the string s may be generated by a single imbricator. If the substring of s not generated by the range dependent replicator cannot be generated by a wide concatenator then s may not be generated by a sequence replicator. We have to point out that the syntax of such replicators has to be expressed by context-sensitive rules if these are to form well-formed basic strings after expansion. Only by context-sensitive rules we can specify that the number of opening or closing parentheses of their context must be equal to the number of regularities the replicator is to generate, determined by the values of "in", "fi" and "inc" of the index specification part of the replicator.

If the range replicator in s is not a wide concatenator but a wide imbricator then s may not in general be generated by a single imbricator. It may however be abbreviated by generating parts of it by more than one sequence replicator. For this to be possible though, it is still necessary for the part of s not generated by the range dependent replicator to be generated by a wide concatenator. We shall discuss this case when considering neighbourhood dependent replicators below, since the string s may be generated by two replicators which are of this form.

We may always though, generate by sequence replicators strings parts of which have been generated by context dependent replicators provided

they always generate more than one regularity. We construct sequence replicators out of them by rearranging parts of the regularities they generate so that parentheses in each regularity match. These sequence replicators cannot generate the complete strings which context dependent replicators generate but only parts of them. Consider for example the following string

(#i:1,n,1[A(i)),(B(i);@])

involving a context dependent replicator which as all context dependent replicators generates the same number of opening and closing parentheses, though not all matching. Since upon expansion the i'th opening parenthesis matches with the (i+1)'th closing parenthesis for i=1,2,...,(n-1), we may modify the regularity to form the sequence replicator:

NR18 #i:1,n-1,1[(B(i);A(i+1)),@]

The strings "A(1));" and ",(B(n)" which are heading and respectively trailing the expansion of the context dependent replicator, and the opening and closing parentheses around the context dependent replicator are not generated by the above concatenator and have to be written explicitly. Thus the string

(A(1));#i:1,n-1,1[(B(i);A(i+1)),@],(B(n))

abbreviates the expansion of the expression involving the context replicator as long as n>1 and it is valid in our notation. If n takes the value 1 the range of the index of the concatenator becomes empty which is not permitted. If n could take the value 1 we may replace the concatenator together with the ";" before or the "," after it by a left or respectively right replicator:

(A(1))#i:1,n-1,1[;|(B(i);A(i+1)),@],(B(n))
(A(1));#i:1,n-1,1[(B(i);A(i+1)),@|,](B(n))

If the context dependent replicator is of the form of imbricators we first transform it into wide concatenators and we then apply, from left

to right, the transformations we outlined above. Consider for example the context dependent imbricator

```
(#i:1,n,1[A(i));(B(i),@@);C(i),(D(i)])
```

Let us transform it first into wide concatenators

```
(#i:1,n,1[A(i);(B(i),@] #i:n,1,-1[);C(i),(D(i)])
```

Then re-arrange the regularity of the leftmost replicator, transforming it into a concatenator

```
(A(1));#i:1,n-1,1[(B(i),A(i+1));@];(B(n)#i:n,1,-1[);C(i),(D(i)])
```

Do the same for the other wide concatenator

```
 (A(1))
;#i:1,n-1,1[(B(i),A(i+1));@]
;(B(n))
;C(n),#i:n-1,1,-1[(D(i));C(i+1),@],(D(1))
```

The above string is valid in our notation. We may simplify it by eliminating a number of redundant parentheses which could not be easily detected in the original expression involving the two neighbourhood dependent replicators:

```
 A(1)
;#i:1,n-1,1[B(i),A(i+1);@]
;B(n)
;C(n),#i:n-1,1,-1[D(i);C(i+1),@],(D(1))
```

We may also replace the two concatenators and the string between them by a non-genuine imbricator thus simplifying the above expression, even further:

```
A(1);#i:1,n-1,1[B(i),A(i+1);@;B(n);C(n),@,D(i);C(i+1)],D(1)
```

Finally, basic sequences generated by groups of neighbourhood dependent replicators may be generated by sequence replicators provided they generate more than one regularity similarly to the context dependent replicators. Let us abbreviate the expansion of

```
#i:1,n,1[(A(i);@@),(B(i)] ; #i:1,n,1[C(i));(@@;D(i))]
```

by sequence replicators. We first split the two replicators into wide concatenators as follows:

```
#i:1,n,1[(A(i);@]#i:n,1,-1[),(B(i)];
#i:1,n,1[C(i));(]#i:n,1,-1[D(i));@]
```

Then re-arrange parts of regularities of these replicators balancing the parentheses

```
#i:1,n-1,1[(A(i);@];
(A(n));#i:n,2,-1[(B(i)),@],(B(1);C(1));#i:2,n,1[(C(i));@];(D(n));
#i:n-1,1,-1[D(i));@]
```

We may now eliminate some redundant parentheses in the above expression, thus obtaining:

```
#i:1,n-1,1[(A(i);@];
A(n);#i:n,2,-1[B(i),@],(B(1);C(1));#i:2,n,1[C(i);@];D(n);
#i:n-1,1,-1[D(i));@]
```

The above expression contains two concatenators and two neighbourhood dependent replicators which may be replaced by one imbricator, as follows:

```
NR19 #i:1,n-1,1
       [( A(i)
         ;@
         ;A(n)
         ;#i:n,2,-1[B(i),],(B(1);C(1))
         ;#i:2,n,1[C(i);]
         ;D(n)
         ;@
         ;D(i)
        )
       ]
```

which may be abbreviated by combining the two concatenators between the two "@"s into one non-genuine imbricator

```
#i:1,n-1,1
 [(A(i);@;A(n);#j:n,2,-1[B(i),@,(B(1);C(1));@;C(i)];D(n);@;D(i))]
```

We may easily verify by expansion that the above imbricator generates the same string as the two original neighbourhood dependent replicators. Although the above string is lenghtier than that involving the neighbourhood dependent replicators its advantages in understanding it compensates for this loss of conciseness. This is true of all three of the types of replicators we eliminated.

d. left and right replicators

These may generate empty expansions or sequences followed or preceeded by a separator. Previous grammars permit replicators which may generate a subclass of this type of strings. Left and right replicators do not only generate more strings of this type but by specifying their context they guarantee the well-formedness of the expanded program.

e. distributors

Their contribution in the notation is not in the generality of the strings they generate since the same strings may be generated by

concatenators but in the conciseness in representing these strings. We have extended the type of regularities they may generate. These regularities may include replicators also, thus making our two macro features symmetrical as one may be nested within each other. We have also relaxed the compatibility criterion for distributable dimensions. In previous notations it was required that these should have the same set of subscripts in these dimensions. We only require that the number of these subscripts are the same.

We have further extended the class of sequences which the distributors may generate by the introduction of two features: the subrange and the dimension selection.

All these extensions greatly improve the conciseness of macro programs since more strings could be generated by distributors in the new notation than in any other macro notation used before.

3. The readability of macro programs in the new notation is also greatly improved. This was mainly achieved by the following:

a. By changing the index specification part of a replicator from "$\boxed{i}$|in,fi,inc" to "#i:in,fi,inc" and moving it in front of "[ ]" of the replicator which now just encloses the regularity to be replicated.

b. By changing "( )" around the regularity of the distributor to "[ ]". By that we have distinguished symbols not used in the basic notation but only in the macro notation. Distributors are now easily identified in a macro sequence.

c. By eliminating range, context and neighbourhood dependent replicators which had to be understood in conjuction with other parts of a macro sequence.

d. By permitting replicators in sequences to generate regularities which are sequences separated by semicolons or commas, which means that their expansion consists of familiar substrings.

We have to point out that this is not the best notation for readability of macro programs. Its weakness is that in general, the head and/or tail of their expansion may bind with strings before or after rather than with the rest of the expansion. In other words their expansion is not syntactically strong, in general. Also each regularity may not be syntactically strong in the expansion.

In chapter 4 we give two grammars which restrict the strings replicators may generate. The first produces replicators and distributors generating syntactically strong expansions and the second produces replicators and distributors which additionally generate syntactically strong regularities. These two grammars greatly improve the readability of macro programs, particularly macro sequences as we shall demonstrate in chapter 4. Of course programs produced by the grammars of chapter 4 are not as concise in general as macro programs produced by the notation of section 3.2.

4. The syntax of the notation of section 3.2 is uniform with that of the basic notation. Basic program bodies have been extended to macro program bodies by permitting collectivisors and bodyreplicators. Basic sequences in paths and processes have been extended to macro sequences by permitting indexed operations and generalized elements which may involve replicators and distributors.

The production rules for macro sequences look very similar in structure as the syntax rules of basic sequences. Also the rules producing the strings inside "[ ]" of sequence replicators and distributors have been expressed in the style of a basic sequence, as "extended" regular expressions.

The notation introduced in section 3.2 could be extended by other features and its existing features could be generalized. We shall discuss one new feature, distributors generating paths and/or processes and two generalizations of existing features, the index specification of replicators not necessarily generating integers forming arithmetic progressions and a more flexible selection of distributable dimensions of distributors.

As there are replicators, the bodyreplicators, generating paths and/or processes we may have "bodydistributors" generating some paths and/or processes more concisely than replicators. There is no problem in principle as bodyreplicators generate consecutive regularities only. The only problem is that there is no connective between paths and processes and therefore there is no connective to be distributed separating the regularities they would generate. We may not use a connective at all in front of bodydistributors. With this convention the n free frame buffer may be specified by

[path DEPOSIT;REMOVE end]

where DEPOSIT and REMOVE have been defined by NC18

NC18 array DEPOSIT REMOVE(n) endarray

When a subrange is incorporated we may write

#1,n,2[path DEPOSIT;REMOVE end]

to specify the odd frames of the n-free-frame buffer.

This form however looks very similar to that of replicators and could effect the readability of the programs. For this reason we were reluctant to include it in the notation but we only mentioned it here as a possible option, for further extensions of the macro notation.

As we have seen the index specification part of replicators generates finite arithmetic progressions of integers in ascending or descending order. This kind of index generation proves to be very powerful in generating indices of subscripted operations. Nevertheless, the index specification of replicators could be extended to generate finite collections of integers not necessarily forming arithmetic progressions. The predicate or test replicator [LS80] (cf. 3.1.9) are examples of replicators using such generators. The predicates is a convenient and powerful tool for generating finite collections of integers. Another way to generate indices is to use generating functions which could be specified in some conventional programming language. The index

specification part as it now stands would be specified by a for-statement. As not enough experimentation has been done to satisfy us for the best way to specify this general generation of index replicator range and as both of the above suggested methods introduce great complexity we did not incorporate them in the notation.

The feature for the selection of distributable dimensions has a limitation, already indicated by condition 5 of theorem 3.10 giving the conditions for the replacement of sequence replicators by distributors. This condition suggests that it is not possible to generate the string generated by the replicators:

```
NR20 #i:1,n,1[(#j:1,m,1[A(j,i);B(i,j);@]),@]
```

where A and B are defined by

```
NC19 array A(m,n) B(n,m) endarray
```

by two nested distributors. We may either replace the inner replicator by a distributor as:

```
ND20 #i:1,n,1[(;[A(,i);B(i,)]),@]
```

or the outer one as

```
ND21 ,[(#j:1,m,1[A(j,);B(,j);@])]
```

The replicators in the above two expressions may not be replaced by distributors in a valid way. The following expression

```
,[(;[A;B])]
```

involving two nested distributors is not valid since the outer distributor applies to the second dimensions of A and B which are not compatible and futhermore is not what the replicators specify.

If however, we allow each dimension to specify the distributor to be applied to it then distributors may replace both the above replicators.

Therefore instead of specifying distributable dimensions by blank fields, we must define them by an integer, specifying the level of the distributor applied to it, which has to be distinguished from subscripts, by, say pefixing it by "#". With this convention we may replace the above nested replicators by

,[(;[A(#2,#1);B(#1,#2)])]

This feature lenghtens distributors substantially and as it is only useful in special cases we did not include it in the new notation.

In this chapter we reviewed previous macro notations and grammars, we introduced a new macro notation and grammar and proved several syntactic properties which macro elements and complete macro programs possess. In the next chapter we address the problem of the semantics of macro programs.

4 THE SEMANTICS OF MACRO COSY PROGRAMS

In the previous chapter we obtained fairly complete results relative to the syntax and expansion of macro programs, but no reference at all was made to their semantics. The theorem 3.11 in section 3.3.3, proving the expansion of macro programs to basic programs allows us to define their semantics in terms of the vector firing sequences of the basic programs obtained by their expansion. The semantics of a macro program MPROG which does not include any macro processes will therefore be given in terms of

VFS(expand(MPROG))

and that of a macro program involving processes in terms of

VFS(Path(expand(MPROG))) or
MVFS(expand(MPROG))

where the conctruction of VFS and MVFS and the transformation Path are defined in chapter 2, and the function "expand" in section 3.3.3.

In this chapter we examine ways by which the vector firing sequences of basic programs generated from macro programs may be obtained directly from the macro programs themselves. We shall restrict our discussion to programs involving just macro paths and bodyreplicators generating macro paths.

We may recall from chapter 2 that to obtain the vector firing sequences of a basic path-program PROG we need two sets:

1. the set of all vectors each component of which is a firing sequence of a path in PROG, and

2. the set of vector operations in the program PROG, the set Vops(PROG).

To construct therefore, the vector firing sequences of the expansion of a macro program from the macro program itself, we need to construct both sets mentioned in 1 and 2 above directly from it. To obtain the set in 1, the cycle set of each basic path generated from a macro program should be constructed from the macro program itself. These cycle sets should be totally ordered and their ordering should be the same as the ordering of their corresponding basic paths in the expanded program, and will be called ordered cycle sets. If the cycle sets obtained from the macro program are the same as the cycle sets obtained from the basic paths in the basic program generated by the expansion of the macro program but their ordering is different, the vector firing sequences produced by these two collections of sets will in general, be different. This order of cycle sets was implicit in the construction of the vector firing sequences of a basic program in section 2.3, being the order of appearance of their corresponding paths in the basic program.

The second set we have to obtain direcly from macro programs, the set of vector operations of corresponding expanded programs Vops(PROG) may be obtained from the ordered cycle sets as it was shown in section 2.3. Assuming that the ordered cycle sets may be obtained directly from macro programs, then so may the set of vector operations Vops, and consequently the vector firing sequences as well.

In the rest of this chapter we concentrate on how we may find the ordered cycle sets of expanded programs directly from the macro programs themselves. We follow two approaches for constructing these sets. According to the first, they are constructed by finding the cycle sets of expanded parts of macro programs which are then juxtaposed, when corresponding to cycle sets of paths, or combined by the concatenation operation, when corresponding to cycle sets of orelements or by the union operation, when corresponding to cycle sets of starelements. According to the second approach, macro cycle objects are constructed from macro programs, representing concisely and generating upon expansion ordered cycle sets, in the same way, macro programs represent and generate basic programs.

In section 4.1 we follow the first approach, giving rules for constructing the ordered cycle sets of macro programs produced by the grammar of section 3.2. We also give rules for obtaining the cycle sets of macro programs produced by a restrictive grammar by which all macro elements generate syntactically strong strings.

In section 4.2 we follow the second approach. This approach however may only be applied to macro programs produced by a more restrictive grammar than that of section 4.1.2 producing macro elements generating regularities which are syntactically strong strings. We first develop this grammar, we give expansion rules for programs produced by it and we outline syntactic properties which macro elements and programs produced by this grammar possess. Then we present a notation for representing ordered cycle sets concisely, we define rules for obtaining objects in this notation from macro programs and we give expansion rules by which these objects generate ordered cycle sets which are shown to be the same as those obtained from expanded macro programs.

4.1 CONSTRUCTING ORDERED CYCLE SETS UPON EXPANSION OF MACRO PROGRAMS

We split the construction of ordered cycle sets into two parts. In the first part ordered expressions for obtaining cycle sets of individual macro paths are derived from macro programs upon expansion of their bodyreplicators. As the expansion of a single macro path is a single basic path, we obtain as many such expressions as basic paths in the expanded program. Furthermore, the order of these expressions will be the same as the order of corresponding basic paths in the basic program obtained by the the expansion of macro programs.

In the second part of the construction of the ordered cycle sets of a basic program PROG generated by the expansion of a macro program MPROG, we obtain cycle sets of single basic paths of PROG directly from corresponding macro paths of MPROG, which after the first part is applied they do not involve any integer expressions involving bodyreplicator indices. We shall call these macro paths the pure macro paths of MPROG.

We construct the ordered cycle sets of macro programs involving only paths and bodyreplicators generating paths, produced by the grammar of section 3.2, and by a restrictive grammar which will be developed in section 4.1.2. The two grammars differ in the way the non-terminal "msequence" is defined and not in any other aspects. The first part of the construction of ordered cycle sets of programs will therefore be common to programs produced by either grammar. The second part in which cycle sets of individual paths produced by the two grammars are obtained, are treated separately in sections 4.1.1 and 4.1.2.

Let us now define the function "exp-Cycls" by which ordered expressions for cycle sets of pure macro paths will be obtained from macro programs. The syntactic variables used in this definition denote the same syntactic entities of macro programs as in the definition of "expand" in section 3.3.3. As no processes are involved in these programs though, we will drop the "Q" from the syntactic variables "CPQBRi" for i=1,...,n, "PQBR", "PQBRs", "PQBRi" for i=1,...,n, which thus become "CPBRi" for i=1,...,n, "PBR", "PBRs", "PBRi" for i=1,...,n respectively. In addition "MP" will denote a single pure macro path.

exp-Cycls(e)=cases e:

1. program MPBODY endprogram → cycles exp-Cycls(MPBODY) endcycles
2. CPBR1...CPBRn → exp-Cycls(CPBR1)&...&exp-Cycls(CPBRn)
3. COLs PBR → exp-Cycls(PBR)
4. #j:1,m,1[PBRs] → exp-Cycls(replexp^{0}(#j:1,m,1[PBRs]))
5. PBR1...PBRn → exp-Cycls(PBR1)&...&exp-Cycls(PBRn)
6. MP → if produced by grammar of section 3.2
   then exp-Cycl(MP)
   else
   if produced by grammar of section 4.1.1
   then exp-Cyc2(MP)

In the above definition the two functions introduced in case 6 "exp-Cycl" and "exp-Cyc2", will be defined in sections 4.1.1 and 4.1.2; they yield the cycle sets of individual pure macro paths, originating from macro programs produced respectively by the grammars of sections 3.2 and 4.1.2. The symbol "&" on the right hand side of cases 2, 4, 5

is used to separate cycle sets.

Let us also define the function "expand1" by which macro programs may be expanded. We have modified slightly the first six cases of the definition of function "expand" of section 3.3.3, adopting the above changes in syntactic variables; the expansion of macro paths is defined by two distinct functions depending on whether these are produced by the syntax rules of section 3.2 or that of 4.1.2. By applying the function "expand1" we therefore obtain expressions for the expansion of individual pure macro paths. The function "expand1" is defined as follows:

expand1(e)=cases e:

1. program MPBODY endprogram → program expand1(MPBODY) endprogram
2. CPBR1...CPBRn → expand1(CPBR1)...expand1(CPBRn)
3. COLs PBR → expand1(PBR)
4. #j:1,m,1[PBRs] → expand1(replexp^0(#j:1,m,1[PBRs]))
5. PBR1...PBRn → expand1(PBR1)...expand1(PBRn)
6. MP → if produced by grammar of section 3.2
   then path-exp1(MP)
   else
   if produced by grammar of section 4.1.1
   then path-exp2(MP)

where path-exp1(MP) denotes the expansion of a pure macro path MP of a macro program produced by the grammar in section 3.2 and path-exp2(MP) denotes the expansion of a pure macro path MP of a macro program produced by the grammar in section 4.1.2.

The similarity of the definitions of the functions "exp-Cycls" and "expand1", shows that there exists an exact correspondence between construction of the cycle sets of macro paths and construction of the expansion of macro paths. Let us define the function "Cycles", by which the ordered cycle sets of basic programs are obtained:

Cycles(e)=cases e:

1. program BPBODY endprogram → cycles Cycles(BPBODY) endcycles
2. P1...Pn → Cycles(P1)&...&Cycles(Pn)
3. P → Cyc(P)

where BPBODY denotes a basic path program body which is represented by

P1...Pn

where Pi for i=1,2,...,n denote basic paths and P denotes a single basic path. The function "Cyc" is defined in section 2.1. We may easily show that for a macro program MPROG, the relation

Cycles(expand1(MPROG)=exp-Cycls(MPROG)

is true, provided that, if MPROG is produced by the grammar of 3.2 then

Cyc(path-exp1(MP))=exp-Cyc1(MP)

for any pure macro path MP of MPROG and that, if MPROG is produced by the grammar of 4.1.2 then

Cyc(path-exp2(MP))=exp-Cyc2(MP)

for any pure macro path MP of MPROG.

The validity of the above two equalities will be proven formally in the next two subsections 4.1.1 and 4.1.2 respectively, where we define the functions "exp-Cyc1" and "exp-Cyc2" by which the cycle set of a single basic path may be obtained directly from its unexpanded pure macro path, and where we also define the functions "path-exp1" and "path-exp2" by which pure macro paths are expanded.

4.1.1 Finding the Cycle Sets of pure macro Paths

In this subsection we define the function "exp-Cycl" by which we obtain the cycle sets of pure macro paths produced by the grammar of section 3.2.

The function "exp-Cycl" expands parts of a macro sequence, constructs the cycle sets of these parts and performs concatenation or union operations on them until the cycle set of the whole path is constructed. What the smallest such parts of macro sequences should be is governed by the syntax of the macro path. The reason for considering some smallest parts is that it only makes sense to find the cycle set of a syntactically strong string or of macro elements generating such strings. Had we allowed range, context and neighbourhood dependent replicators in macro sequences we would in general, have to expand the whole of a macro sequence, to construct the cycle set of a macro path which involved such sequences. Consider for example the paths P1, P2, P3 the macro sequences of which involve range, context and neighbourhood dependent replicators, respectively:

P1 path ((b,#i:1,2,1[A(i));c,@] end
P2 path (c;#i:1,2,1[A(i));(B(i);@]) end
P3 path #i:1,3,1[(UP(i);@];#i:3,1,-1[DOWN(i))*;@] end

which expand respectively to P4, P5 and P6:

P4 path ((b,A(1));c,A(2));c end
P5 path c;(A(1)),(B(1);A(2)),(B(2)) end
P6 path(UP(1);(UP(2);(UP(3);DOWN(3))*;DOWN(2))*;DOWN(1))* end

We cannot find the cycle sets of any parts of the macro sequences of P1, P2, P3, since the replicators they involve do not generate matching opening and closing parentheses and the precedence of connectives "," and ";" in their context may be overuled by the generation of parentheses upon the expansion of the range, context and neighbourhood dependent replicators, thus making it impossible to detect the syntactically strong strings without expanding completely the macro sequences they are in.

The macro paths produced by the grammar of section 3.2 are such that we may break up their macro sequences into their macro orelements the cycle sets of which may be constructed and which may then be concatenated to give the cycle set of the complete macro path. The string generated by the expansion of the macro elements produced by this grammar is always syntactically strong in the context of any of "path", ";", "(" on their left and any of ")", ";", "end" on their right. The reason is that the precedence of "," over ";" cannot be overuled by the expansion of these macro elements produced by this grammar since the macro elements always generate macro sequences and consequently matching pairs of parentheses. Therefore, to find the cycle set of a macro sequence, we may concatenate the cycle sets of their constituent orelements. If these orelements involve only starelements, we construct their cycle sets by the union of the cycle sets of these starelements. If however, the orelements involve generalized elements, all replicators and distributors which they involve have to have been expanded first, as they may generate semicolons which would transform the original macro orelement into a macro sequence. Let us define an auxilliary function "gel-exp" by which all replicators and distributors of a generalized element are expanded. If we represent a generalized element GEL by

RR1...RRn M LR1...LRm

where each of RRi for i=1,...,n is a right replicator, each of LRi for i=1,...,m is a left replicator and M a sequence replicator or a distributor or a starelement, then by the expansion of GEL denoted by gel-exp(GEL), we mean the string obtained by the expansion of the right replicators, the expansion of M if its a sequence replicator or a distributor, and by the expansion of the left replicators. The function "gel-exp" is defined by: gel-exp(GEL)=a b c

where a= $replexp^0$(RR1)...$replexp^0$(RRn)
b= if M is a sequence replicator then $replexp^0$(M) else
if M is a distributor then $distrexp^0$(M) else M
c= $replexp^0$(LR1)...$replexp^0$(LRm)

If M is a starelement the function "gel-exp" is the same as "$gelexp^0$", defined in section 3.3, the section in which "$replexp^0$" and "$distrexp^0$"

have also been defined.

We shall use this function when defining "path-expl", by which macro paths are expanded. The difference between "path-expl" and corresponding cases 6 and 8 to 15 in the function "expand" of section 3.3.3 is that when "path-expl" is applied to a syntactic entity S, it will not always distribute over the syntactic subentities of S but only if they are syntactically strong strings or macro elements generating such strings. In particular, the expansion of a macro orelement consisting of generalized elements will not be defined by juxtaposition of the expansions of its constituent generalized elements separated by commas, as in general, the expansion of generalized elements are macro sequences which are not syntactically strong in the context of a comma on their left or their right. The expansion of a macro orelement will be defined as the expansion of the string, sequence in general, obtained after the function "gel-exp" is applied to all its generalized elements. When however, a macro orelement consists entirely of starelements the its expansion will be defined by the juxtaposition of the expansions of its constituent starelements separated by commas. Syntactic entities corresponding to macro sequences, orelements, generalized elements, starelements, elements and operations will be represented by MSEQ, MORi for i=1,...,n, GELi for i=1,...,n, STELi for i=1,...,n, EL and OP respectively. Formally the function "path-expl" is defined by:

path-expl(e)=cases e:

1. path MSEQ end → path path-expl(MSEQ) end
2. MOR1;...;MORn → path-expl(MOR1);...;path-expl(MORn)
3. GEL1,...,GELn → path-expl(gel-exp(GEL1),...,gel-exp(GELn))
4. STEL1,...,STELn → path-expl(STEL1),..., path-expl(STELn)
5. EL* → path-expl(EL)*
6. (MSEQ) → (path-expl(MSEQ))
7. OP → OP + possible expression evaluations

We shall not formally prove that the expansion of a macro path P, path-expl(P) yields a basic path but we only point out that it may be proven in the style of theorem 3.11 in section 3.3.3.

Let us now formally define the function "exp-Cycl" by which the cycle set of an expanded path may be obtained directly from the macro path. The function "exp-Cycl" will apply to the same syntactic entities as the function "path-expl" above.

exp-Cycl(e)=cases e:

1. path MSEQ end → exp-Cycl(MSEQ)
2. MOR1;...;MORn → exp-Cycl(MOR1) ⊙...⊙ exp-Cycl(MORn)
3. GEL1,...,GELn → exp-Cycl(gel-exp(GEL1),...,gel-exp(GELn))
4. STEL1,...,STELn → exp-Cycl(STEL1) U...U exp-Cycl(STELn)
5. EL* → exp-Cycl(EL)*
6. (MSEQ) → exp-Cycl(MSEQ)
7. OP → {OP}

Let us find the cycle set of path P7

P7 path a,#i:1,3,1[B(i);@],c;d end

by applying the function "exp-Cycl":

exp-Cycl(P7)=exp-Cycl(a,#i:1,3,1[B(i);@],c;d)
=exp-Cycl(a,#i:1,3,1[B(i);@],c)⊙exp-Cycl(d)

exp-Cycl(a,#i:1,3,1[B(i);@],c)=
exp-Cycl(a,gel-exp(#i:1,3,1[B(i);@]),c)=
exp-Cycl(a,B(1);B(2);B(3),c)=
exp-Cycl(a,B(1))⊙exp-Cycl(B(2))⊙exp-Cycl(B(3),c)=
{a,B(1)}⊙{B(2)}⊙{B(3),c}=
{a.B(2).B(3),a.B(2).c,B(1).B(2).B(3),B(1).B(2).c}

exp-Cycl(d)={d}

Thus,
exp-Cycl(P7)={ a.B(2).B(3).d, a.B(2).c.d,
B(1).B(2).B(3).d, B(1).B(2).c.d}

The same cycle set may be obtained from the expansion of P7, path P8

P8 path a,B(1);B(2);B(3),c;d end

by applying the function "Cyc" of chapter 2.

We may formally prove the theorem 4.1 for the direct construction of cycle sets from pure macro paths.

THEOREM 4.1:

The cycle set of any pure macro path MP of a macro program produced by the syntax rules in section 3.2 obtained by exp-Cycl(MP) is the same as the cycle set of the basic path obtained by its expansion, or formally

exp-Cycl(MP)=Cyc(path-expl(MP))

Proof:

We shall prove the theorem by considering separately each syntactic case for which "exp-Cycl" defined comparing the results with corresponding results obtained by applying the function "path-expl" and then "Cyc".

case 1

Applying "exp-Cycl" to a macro path we obtain

exp-Cycl(path MSEQ end)=exp-Cycl(MSEQ)

and applying the function "path-expl" and its result to "Cyc" we obtain

Cyc(path-expl(path MSEQ end))=
Cyc(path path-expl(MSEQ) end)=
Cyc(path-expl(MSEQ))

The two results are the same as may be shown by case 2.

case 2

Applying "exp-Cycl" to a macro sequence we obtain

$$\text{exp-Cycl}(MOR1;\ldots;MORn)=\text{exp-Cycl}(MOR1)\bullet\ldots\bullet\text{exp-Cyc}(MORn)$$

and the functions "path-expl" and then "Cyc" we obtain

$$\text{Cyc}(\text{path-expl}(MOR1;\ldots;MORn))=$$
$$\text{Cyc}(\text{path-expl}(MOR1);\ldots;\text{path-expl}(MORn))$$

Since path-expl(MORi) for i=1,...,n yields a basic sequence in general, the above expression is the same as:

$$\text{Cyc}(\text{path-expl}(MOR1))\bullet\ldots\bullet\text{Cyc}(\text{path-expl}(MORn))$$

The last step is valid since each of path-expl(MORi) for i=1,...,n is a basic sequence and if SEQ1 and SEQ2 are basic sequences then

$$\text{Cyc}(SEQ1)\bullet\text{Cyc}(SEQ2)=\text{Cyc}(SEQ1;SEQ2)$$

To show the above relation let

$$SEQ1=OR1^1;\ldots;ORk^1 \quad \text{and}$$
$$SEQ2=OR1^2;\ldots;ORm^2$$

where ORj^1 for j=1,...,n and ORi^2 for i=1,...,m are basic orelements. Then,

$$\text{Cyc}(SEQ1)=\text{Cyc}(OR1^1;\ldots;ORk^1)=$$
$$\text{Cyc}(OR1^1)\bullet\ldots\bullet\text{Cyc}(ORk^1) \quad \text{and}$$

$$\text{Cyc}(SEQ2)=\text{Cyc}(OR1^2;\ldots;ORm^2)=$$
$$\text{Cyc}(OR1^2)\bullet\ldots\bullet\text{Cyc}(ORm^2)$$

Therefore,

$$Cyc(SEQ1)\circ Cyc(SEQ2)=Cyc(OR1^1)\circ\ldots\circ Cyc(ORk^1)\circ Cyc(OR1^2)\circ\ldots\circ Cyc(ORm^2)$$

which is the same as

$$Cyc(SEQ1;SEQ2)=Cyc(OR1^1;\ldots;ORk^1;OR1^2;\ldots;ORm^2)=$$
$$Cyc(OR1^1)\circ\ldots\circ Cyc(ORk^1)\circ Cyc(OR1^2)\circ\ldots\circ Cyc(ORm^2)$$

Therefore, if for any macro orelement MOR the relation

exp-Cycl(MOR)=Cyc(path-expl(MOR))

holds, then the theorem holds for case 2. The above relation may be shown to be true by cases 3 or 4, depending on whether MOR involves generalized elements or just starelements.

case 3

Applying "exp-Cycl" to a macro orelement involving generalized elements we obtain

exp-Cycl(GEL1,...,GELn)=
exp-Cycl(gel-exp(GEL1),...,gel-exp(GELn))

and applying the functions "path-expl" and then "Cyc" we obtain

Cyc(path-expl(GEL1,...,GELn))=
Cyc(path-expl(gel-exp(GEL1),...,gel-exp(GELn))

The expression

gel-exp(GEL1),...,gel-exp(GELn)

is a macro sequence in general since each of gel-exp(GELi) for i=1,...,n is a macro sequence in general, by lemma 3 of section 3.3.1. Therefore, the equality of the above expressions may be shown by case 2.

case 4

Applying "exp-Cycl" to an orelement consisting entirely of starelements we obtain

exp-Cycl(STEL1,...,STELn)=exp-Cycl(STEL1)U...U exp-Cycl(STELn)

and applying "path-expl" and then "Cyc" we obtain

Cyc(path-expl(STEL1,...,STELn))=
Cyc(path-expl(STEL1),...,path-expl(STELn))=
Cyc(path-expl(STEL1))U...U Cyc(path-expl(STELn))

Therefore, if for any starelement STEL the relation

exp-Cycl(STEL)=Cyc(path-expl(STEL))

holds, then the theorem holds for case 4. The above relation may be shown to be true by case 5.

case 5

Applying "exp-Cycl" to a starelement we obtain

exp-Cycl(EL*)=exp-Cycl(EL)*

and applying "path-expl" and then "Cyc" we obtain

Cyc(path-expl(EL*))=Cyc(path-expl(EL)*)=Cyc(path-expl(EL))*

The equality of the two expressions may be shown by case 6.

case 6

Applying "exp-Cycl" to an element of the form (MSEQ) we obtain

exp-Cycl((MSEQ))=exp-Cyc(MSEQ)

and by applying "path-expl" and then "Cyc" we obtain

Cyc(path-expl((MSEQ)))=Cyc((path-expl(MSEQ)))=Cyc(path-expl(MSEQ))

the equality of which may be shown by case 2.

case 7

Applying "exp-Cycl" to an operation we obtain

exp-Cycl(OP)={OP}

and applying "path-expl" and then "Cyc" we obtain

Cyc(path-expl(OP))=Cyc(OP)={OP}

and as both expressions are the same the theorem is proven.√√√

As we have pointed out the grammar of section 3.2 does not in general produce replicators which generate syntactically strong strings in all the contexts they appear. This occurs when one of the separators on their left or their right is "," and the main connective of the expansion is ";", as indeed may be seen in path P7. Consequently, it would be wrong to construct the cycle set of a macro orelement by constructing the union of the cycle sets of its constituent generalized elements. If we define a function "exp-Cycl'" identical to "exp-Cycl" except for cases 3 and 4 which are replaced by

GEL1,...,GELn → exp-Cycl'(GEL1)U...U exp-Cycl'(GELn)

and apply it to path P7, we obtain:

exp-Cycl'(P7)=exp-Cycl'(a,#i:1,3,1[B(i);@])•exp-Cycl'(d)

exp-Cycl'(a,#1,3,1[B(i);@],c)=
exp-Cycl'(a) U exp-Cycl'(#i:1,3,1[B(i);@]) U exp-Cycl'(c)=
exp-Cycl'(a) U exp-Cycl'(gelexp(#i:1,3,1[B(i);@])) U exp-Cycl'(c)=
exp-Cycl'(a) U exp-Cycl'(B(1);B(2);B(3)) U exp-Cycl'(c)=
{a} U {B(1).B(2).B(3)} U {c}

Therefore exp-Cycl'(P7) yields

{a,B(1).B(2).B(3),c}•{d}=
{a.d,B(1).d,B(2).d,B(3).d,c.d}

which of course is not the cycle set of P7 but of P9

P9 path a,(#i:1,3,1[B(i);@]),c;d end

The reason the above construction failed is that we used the equality

Cyc(A,B)=Cyc(A)U Cyc(B)

which in general is not true unless A and B are orelements which means that A and B are syntactically strong in the whole of "A,B".

To be able to find the correct cycle set of a macro path by the above method, all replicators and distributors should generate syntactically strong strings in any context they appear. In the next subsection we develop syntax rules for the production of restricted macro paths involving only such replicators and distributors and define the function "path-exp2" by which these are expanded. We also define the function "exp-Cyc2" by which the cycle sets of pure macro paths of programs produced by this grammar may be constructed directly from them.

4.1.2 Finding the Cycle Sets of Restricted pure macro Paths

In the grammar in this subsection the syntax rules for macro sequences will be modified. Left and right replicators will be eliminated and the rest of macro elements generating macro sequences,

macro orelements, macro starelements and macro elements will be produced by distinct syntax rules guaranteing that their expansion is always syntactically strong. In the syntax rules in this section we follow the same meta-language conventions as in chapter 3.

Macro elements generating macro sequences will be permitted to appear only between any of "path", ";", "(" on their left and any of "end", ";", ")" on their right.

The new production rule for "msequence" is:

msequence={seqpart ③;}+

seqpart=seqmacro/morelement

where "seqpart" produces parts of macro sequences separated by ";" which may be either macro elements strictly generating macro sequences, produced by "seqmacro" or macro orelements, produced by "morelement".

The new rules for "morelement" are:

morelement={orpart ②,}+

where "orpart" denotes parts of macro orelements separated by ",". These parts may be macro elements strictly generating orelements, produced by "ormacro", or macro elements strictly generating starelements, produced by "starmacro"; they could also be starred elements, produced by "mstarelement". The latter is prefixed by "m" as we permit certain macro elements to be starred. The syntax of "orpart" is given by:

orpart=ormacro/starmacro/mstarelement

The non-terminal "mstarelement" produces elements which could be starred as can be seen in the following rule:

mstarelement=element/element*

where the non-terminal "element" is defined by:

```
element=indexedop/operation/(msequence)/elmacro
```

where the non-terminal "elmacro" produces macro elements which generate elements. The syntax for "seqmacro" is:

```
seqmacro=seqrepl/seqdistr
```

where "seqrepl" and "seqdistr" produce replicators and distributors respectively, generating strictly macro sequences, and will be called strict sequence replicators and strict sequence distributors respectively. Strict sequence replicators could either be concatenators or imbricators. The syntax for "seqrepl" will be defined by:

```
seqrepl=index_spec[{seqconcseq/seqimbrseq}]
```

where "index_spec" has been defined in section 3.2, "seqconcseq" and "seqimbrseq" denote strings inside "[ ]" of strict sequence concatenators and imbricators respectively. For strict sequence concatenators and distributors to generate strictly sequences, either the main connective of their regularities should be a ";", or their regularities should be separated by ";". The syntax of "seqconcseq" and of "seqdistr" will therefore be defined by:

```
seqconcseq={seqpart;}+ {@/seqconcor}

seqconcor={orpart ,}+ @

seqdistr=;{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]
        /,{/iexpr}{/#iexpr,iexpr,iexpr}[{seqpart{;seqpart}+]
```

As in the distributors of section 3.2 the "operations" produced by "msequence" and "seqpart" in the above rule, will be array slices (cf. section 3.2.2).

Strict sequence imbricators may be either genuine or not. As in either case they should strictly generate sequences, the main connective

of the whole expansion should be a semicolon which implies that the main connective of the string produced by "seqimbrseq" should also be a semicolon. The syntax rule for "seqimbrseq" is:

```
seqimbrseq=seqimbr_atout_seq
          /{seqpart ;}+ seqimbror {; seqpart}*
          /{seqpart ;}* seqimbror {; seqpart}+
```

The non-terminal "seqimbr_atout_seq" produces the string inside "[ ]" of a non-genuine imbricator, and its syntax may be obtained from the syntax of "imbr_at_seq" of section 3.2 exluding productions which do not produce at least one ";". This implies that "seqimbr_atout_seq" may produce strings which are produced by the alternative productions 1, 2, 3, 5, 6, 7, 10, 12 and 13 of "imbr_at_seq". The occurrences of the non-terminal "morelement" in these rules should be replaced by the non-terminal "seqpart". The complete rules may be found in appendix C.

The second and third alternative productions for "seqimbrseq" guarrantee that at least one ";" is produced in the string inside "[ ]" of a genuine imbricator. The syntax for "seqimbror" is given by:

```
seqimbror={orpart ,}* seqimbrstarel {, orpart}*

seqimbrstarel=seqimbrel/seqimbrel*

seqimbrel=({seqimbr_atin_seq/seqimbr_in_seq})
```

In the last rule the non-terminal "seqimbr_atin_seq" produces strings which involve the "@"s and "seqimbr_in_seq" strings which involve the "@"s but nested within "( )". As the main connective of the string inside "[ ]" of a strict sequence imbricator is already specified to be a ";", these non-terminals may produce strings which may not involve the ";". The syntax for "seqimbr_in_seq" is given by:

```
seqimbr_in_seq={seqpart ;}* seqimbror {; seqpart}*
```

The syntax for "seqimbr_atin_seq" will be the same as the syntax for "imbr_at_seq" of section 3.2, with all the occurrences of "morelement"

replaced by "seqpart". Again the complete definition for "seqimbr_atin_seq" may be found in appendix C.

The macro elements strictly generating orelements could either be strict orelement replicators or strict orelement distributors produced by "orrepl" and "ordistr" respectively. The syntax of "ormacro" is given by:

```
ormacro=orrepl/ordistr
```

where the non-terminal "orrepl" produces strict orelement replicators and "ordistr" strict orelement distributors. The definition of "ordistr" is:

```
ordistr=,{/iexpr}{/#iexpr,iexpr,iexpr}[morelement]
```

As in all syntax rules for distributors the "operations" in the string produced by "morelement" are array slices (cf. section 3.2).

Strict orelement replicators may be either concatenators or imbricators, the string inside "[ ]" of which is produced by "orconcor" and "orimbror" respectively:

```
orrepl=index_spec[{orconcor/orimbror}]
```

For strict orelement concatenators to generate strictly orelements the main connective in each regularity and the connective separating regularities should be ",". The syntax of "orconcor" is given by:

```
orconcor={orpart ,}+@
```

For imbricators to generate strictly orelements the main connective of its expansion should be "," which implies that the main connective of the string inside "[ ]" should be "," also. The syntax for "orimbror" is given by:

```
orimbror=orimbr_atout_or
        /{orpart,}+ orimbrstarel {,orpart}*
        /{orpart,}* orimbrstarel {,orpart}+
```

where "orimbr_atout_or" produces the string inside "[ ]" of a non-genuine strict orelement imbricator. The main connective of the string it produces should be ",". Its syntax may be obtained from the alternative productions of "imbr_at_seq" of section 3.2 which do not produce a ";", and is given by:

```
orimbr_atout_or=@ {at_orlf/at_orlm}
                /{at_orlm/at_orlb} @
                /at_or2mm
                /@ morelement @
```

The second and third alternative productions for "orimbror" guarantee that at least one "," and no ";" is produced as the main connective of the string inside "[ ]" of a genuine imbricator strictly generating orelements. The syntax of the non-terminal "orimbrstarel" is given by:

```
orimbrstarel=orimbrel/orimbrel*

orimbrel=(orimbrseq)
```

As the main connective of the string inside "[ ]" is a comma, the main connective of the string generated by "orimbrseq" could be ";", as it is nested within "( )" and consequently the ";" cannot be the main connective of the string inside "[ ]". Its syntax is given by:

```
orimbrseq={seqpart ;}* orimbr_in_or {; seqpart}*
        /orimbr_atin_seq
```

The non-terminal "orimbr_atin_seq" produces strings which involve "@"s. As these strings are nested within "( )" their main connective may be a ";". These strings however, cannot be as general as the strings generated by "seqimbr_atin_seq" above. According to the expansion rule for replicators (cf. section 3.2.1) the expansion of imbricators is still defined when their index range is empty provided the string

between the two "@"s with its leading and trailing separators removed is non-empty. For imbricators produced by "ormacro" to generate orelements for any legal range of their indices this string must be a macro orelement. Therefore, from the alternative production rules for "seqimbr_atin_seq" we shall eliminate those which produce ";" between the two "@"s. Because the correct rules are lengthy we give them in appendix C.

The non-terminal "orimbr_in_or" produces strings which nest the "@ t @" further. Its syntax is given by:

orimbr_in_or={orpart ,}* orimbrstarel {, orpart}*

The non-terminals "at_or1f", "at_or1m", "at_or1b", "at_or2mm" "at_or2fm", "at_or2fb", "at_or2mb" in the syntax rules for "seqimbr_atin_seq" and "seqimbr_atout_seq" are obtained from corresponding ones in section 3.2 by replacing all occurrences of "gelement" by "orpart". Their complete rules may be found in appendix C.

The syntax rule for "starmacro" produces macro elements strictly generating starelements of the form:

(msequence)*

As concatenators only generate such strings when they generate a single regularity, "starmacro" will only produce imbricators, called strict starelement imbricators. Furthermore, they will always be genuine. The syntax for "starmacro" is:

starmacro=index_spec[(starimbrseq)*]

where "index_spec" has been defined in section 3.2. The syntax rule for "starimbrseq" may produce sequences, almost as general as the the rule for "seqimbr" in section 3.2. However, certain strings produced by alternative productions for "seqimbr" have to be excluded: those in which the string between the two "@"s with its leading and terminating separators removed form sequences or orelements. The whole string

produced by "starimbrseq" though, may be a sequence or an orelement, as the following rules show:

```
starimbrseq=starimbr_at_seq
          /{seqpart ;}* starimbror {; seqpart}*

starimbror={orpart ,}* starimbrstarel {, orpart}*

starimbrstarel=starimbrel/starimbrel*

starimbrel=(starimbrseq)
```

The syntax rule for "starimbr_at_seq" produces sequences which involve the "@"s. The precise rule for it may be found in appendix C. Similarly to the syntax rule for "starmacro", the syntax rule for "elmacro" may only produce genuine imbricators, called strict element imbricators. Their syntax is given by:

```
elmacro=index_spec[(elimbrseq)]
```

where "index_spec" is defined in section 3.2. The syntax rule for "elimbrseq" is very similar to "starimbrseq". Their only difference is that, if the string between the two "@"s with the leading and terminating separators removed is not null then, if the string inside "[(...)]" is produced by "starimbrseq" is also produced by "mstarelement", but if produced by "elimbrseq" it may be produced by "element". The precise rules may be found in appendix C.

Every replicator and distributor produced by the above rules may be produced by the rules of the grammar of section 3.2. The same though is not true for the context of strict element imbricators which, unlike the replicators of section 3.2, may be starred. If strict element imbricators could not be starred the grammar of section 3.2 would be a true extension of the above. Here we permitted these replicators to be starred since they always generate elements when expanded and the star applies to the sole element generated from the expansion.

Since all replicators and distributors may be produced by the grammar of section 3.2 the same rules for their expansion given in terms of "replexp^0" and "distrexp^0" respectively, will still apply.

We may characterize the expansion of the macro elements produced by the grammar of this subsection as we did for replicators and distributors in section 3.3. Since all macro elements may be produced by the syntax of 3.2 we may use the theorems 3.3 and 3.4 of section 3.3 for the expansion of concatenators and imbricators to macro sequences and the corollary of theorem 3.9 of the same section showing a similar result for the expansion of distributors.

Strict sequence macro elements generate macro sequences in general. When they generate more than one regularity the main connective of the expansion is a semicolon. This is not true though, in general, when they generate one regularity or, in the case of imbricators, their index range is empty, in which case they may generate a single orelement, or starelement, or element. Let us consider the concatenator

#i:1,n,1[A(i);@]

which for n>1 generates sequences. But for n=1 it generates a single element. Let us also consider the non-genuine imbricator

#i:1,n,1[A(i);@,c,@;B(i)]

which for n>1 it generates sequences. For n=1 it generates the orelement

A(1),c,B(1)

and for n=0 the element

c

Genuine imbricators always generate sequences for any non empty range, as the string "@ t @" is nested inside parentheses and the ";", which is the main connective of the string inside "[ ]", is not stripped. But

when the range is empty, they too may generate orelements, starelements or elements. Whatever they generate though, is syntactically strong in their context.

Strict orelement macro elements generate macro orelements. Similarly to the strict sequence macro elements, they may also generate starelements and elements. Strict sequence concatenators and distributors may generate starelements and elements when they generate only one regularity. Strict sequence imbricators, whether genuine or not, may also generate such strings when their index range is empty. As strict orelement macro elements cannot generate sequences their expansion will always be syntactically strong in their context.

Strict starelement imbricators generate starelements except when their index range is empty, in which case they may generate elements.

Finally, strict element imbricators generate elements for any valid range of their indices.

Let us now give some examples of macro paths produced by the above rules.

P10 path f;#i:1,3,1[A(i);B(i),@];,[D],e end
P11 path ;[B,C];,[B;D];,[C,D] end
P12 path #i:1,3,1[(UP(i);@;full*;@;DOWN(i))*],empty end

where collective names A, B, C, D, UP, DOWN are defined by

array A B C D UP DOWN(3) endarray

Let us define the function "path-exp2" for the expansion of pure macro paths of macro programs consisting of macro paths the macro sequences of which are produced by the rules in this section.

Apart from the syntactic variables MSEQ, OP which we have used before denoting macro sequences and operations respectively, we need to introduce some new ones. The syntactic variables SEQPRTi for i=1,...,n

denote either macro orelements or strict sequence macro elements, SEQREPL and SEQDISTR denote strict sequence replicators and distributors respectively. The syntactic variables ORPRTi for i=1,...,n denote macro orelements or macro starelements or starelements; ORREPL and ORDISTR denote strict orelement replicators and distributors respectively. STARMACRO and ELMACRO denote strict starelement and strict element imbricators respectively. Finally, EL denotes elements which could be starred. The function "path-exp2" may be defined by:

path-exp2(e)=cases e:

1.	path MSEQ end	→ path path-exp2(MSEQ) end
2.	SEQPRT1;...;SEQPRTn	→ path-exp2(SEQPRT1);...;path-exp2(SEQPRTn)
3.	SEQREPL	→ path-exp2(replexp^0(SEQREPL))
4.	SEQDISTR	→ path-exp2(distrexp^0(SEQDISTR))
5.	ORPRT1,...,ORPRTn	→ path-exp2(ORPRT1),...,path-exp2(ORPRTn)
6.	ORREPL	→ path-exp2(replexp^0(ORREPL))
7.	ORDISTR	→ path-exp2(distrexp^0(ORDISTR))
8.	STARMACRO	→ path-exp2(replexp^0(STARMACRO))
9.	EL*	→ path-exp2(EL)*
10.	OP	→ OP + possible expression evaluations
11.	(MSEQ)	→ (path-exp2(MSEQ))
12.	ELMACRO	→ path-exp2(replexp^0(ELMACRO))

Using similar arguments to those of theorem 3.11 for the expansion of macro programs to basic programs, we may show that macro programs, the macro paths of which are produced by the syntax rules of this section, also generate basic programs. Let us apply the function "path-exp2" to expand path P10:

path-exp2(P10)= path path-exp2(f;#i:1,3,1[A(i);B(i),@];,[D],e) end

path-exp2(f;#i:1,3,1[A(i);B(i),@];,[D],e)=
path-exp2(f);path-exp2(#i:1,3,1[A(i);B(i),@]);path-exp2(,[D],e)

path-exp2(f)=f

path-exp2(#i:1,3,1[A(i);B(i),@])=
path-exp2(replexp^0(#i:1,3,1[A(i);B(i),@]))=
path-exp2(A(1);B(1),A(2);B(2),A(3);B(3))=
A(1);B(1),A(2);B(2),A(3);B(3)

path-exp2(,[D],e)=path-exp2(,[D]),path-exp2(e)

path-exp2(,[D])=path-exp2(distrexp^0(,[D]))=
path-exp2(D(1),D(2),D(3))=
D(1),D(2),D(3)

path-exp2(e)=e

Therefore, the expansion of P10 is

path f;A(1);B(1),A(2);B(2),A(3);B(3);D(1),D(2),D(3),e end

To obtain the cycle sets of pure macro paths the sequences of which are produced by the syntax rules of this section we define the function "exp-Cyc2" as follows:

exp-Cyc2(e)=cases e:

1. path MSEQ end → exp-Cyc2(MSEQ)
2. SEQPRT1;...;SEQPRTn → exp-Cyc2(SEQPRT1) ∘...∘ exp-Cyc2(SEQPRTn)
3. SEQREPL → exp-Cyc2(replexp^0(SEQREPL))
4. SEQDISTR → exp-Cyc2(distrexp^0(SEQDISTR))
5. ORPRT1,...,ORPRTn → exp-Cyc2(ORPRT1) U...U exp-Cyc2(ORPRTn)
6. ORREPL → exp-Cyc2(replexp^0(ORREPL))
7. ORDISTR → exp-Cyc2(distrexp^0(ORDISTR))
8. STARMACRO → exp-Cyc2(replexp^0(STARMACRO))
9. EL* → exp-Cyc2(EL)*
10. OP → {OP}
11. (MSEQ) → exp-Cyc2(MSEQ)
12. ELMACRO → exp-Cyc2(replexp^0(ELMACRO))

Let us find the cycle set of path P10, by applying "exp-Cyc2":

exp-Cyc2(P10)=
exp-Cyc2(f)•exp-Cyc2(#i:1,3,1[A(i);B(i),@])•exp-Cyc2(,[D],e)

exp-Cyc2(f)={f}

exp-Cyc2(#i:1,3,1[A(i);B(i),@])=
exp-Cyc2(replexp^0(#i:1,3,1[A(i);B(i),@]))=
exp-Cyc2(A(1);B(1),A(2);B(2),A(3);B(3))=
exp-Cyc2(A(1))
•exp-Cyc2(B(1),A(2))
•exp-Cyc2(B(4),A(3))
•exp-Cyc(B(3))=
{A(1)}•{B(1),A(2)}•{B(2),A(3)}•{B(3)}=
{A(1).B(1).B(2).B(3),A(1).B(1).A(3).B(3),
A(1).A(2).B(2).B(3),A(1).A(2).A(3).B(3)}

exp-Cyc2(,[D],e)=
exp-Cyc2(,[D])U exp-Cyc2(e)=
exp-Cyc2(distrexp^0(,[D]))U exp-Cyc2(e)=
exp-Cyc2(D(1),D(2),D(3))U exp-Cyc2(e)=
{D(1),D(2),D(3)} U {e}=
{D(1),D(2),D(3),e}

Therefore,

exp-Cyc2(P10)={f}•{A(1).B(1).B(2).B(3),
A(1).B(1).A(3).B(3),
A(1).A(2).B(2).B(3),
A(1).A(2).A(3).B(3)}•{D(1),D(2),D(3),e}

We may now prove the theorem for the direct construction of cycle sets of pure macro paths.

THEOREM 4.2:

The cycle set of any pure macro path MP the sequence of which is

produced by the grammar of subsection 4.1.2, is the same as the cycle set of the basic path generated by the expansion of MP, or formally

Cyc(path-exp2(MP))=exp-Cyc2(MP)

Proof:

We shall prove the above equality by considering separately each case of syntactic entities on which "exp-Cyc2" and "path-exp2" apply.

case 1

Applying fuction "path-exp2" and then "Cyc" to a macro path we obtain

Cyc(path-exp2(path MSEQ end))=
Cyc(path path-exp2(MSEQ) end)=
Cyc(path-exp2(MSEQ))

and applying "exp-Cyc2" we obtain

exp-Cyc2(path MSEQ end)=exp-Cyc2(MSEQ)

The equality of the above expressions may be shown by case 2.

case 2

Applying "path-exp2" and then "Cyc" to a macro sequence we obtain

Cyc(path-exp2(SEQPRT1;...;SEQPRTn))=
Cyc(path-exp2(SEQPRT1);...;path-exp2(SEQPRTn))

which as we have shown in case 2 of theorem 4.1 is equal to

Cyc(path-exp2(SEQPRT1))•...•Cyc(path-exp2(SEQPRTn))

as each path-exp2(SEQPRTi) for i=1,...,n is a basic sequence. Applying

function "exp-Cyc2" we obtain we obtain

exp-Cyc2(SEQPRT1;...;SEQPRTn)=
exp-Cyc2(SEQPRT1)∘...∘exp-Cyc2(SEQPRTn)

The two expressions are the same provided that for any macro orelement or strict sequence macro element denoted by SEQPRT

Cyc(path-exp2(SEQPRT))=exp-Cyc2(SEQPRT)

holds, which may be shown by cases 3, 4, 5, depending on whether SEQPRT is a macro orelement, a strict sequence replicator, or a strict sequence distributor respectively.

case 3

Applying "path-exp2" and then "Cyc" to a strict sequence replicator we obtain

Cyc(path-exp2(SEQREPL))=Cyc(path-exp2($replexp^0$(SEQREPL)))

and applying "exp-Cyc2" we obtain

exp-Cyc2(SEQREPL)=exp-Cyc2($replexp^0$(SEQREPL))

Since $replexp^0$(SEQREPL) yields a macro sequence the equality of the above expressions may be shown by the previously considered case 2.

case 4

Applying "path-exp2" and then "Cyc" to a strict sequence distributor we obtain

Cyc(path-exp2(SEQDISTR))=Cyc(path-exp2($replexp^0$(SEQDISTR)))

and applying "exp-Cyc2" we obtain

exp-Cyc2(SEQDISTR)=exp-Cyc2(replexp^0(SEQDISTR))

Since replexp^0(SEQDISTR) yields a macro sequence the equality of the above expressions may be shown by the previously considered case 2.

case 5

Applying "path-exp2" and then "exp-Cyc2" to a macro orelement we obtain

Cyc(path-exp2(ORPRT1,...,ORPRTn))=
Cyc(path-exp2(ORPRT1),...,path-exp2(ORPRTn))

Since path-exp2(ORPRTi) for i=1,...,n yields an orelement, the above expression is the same as

Cyc(path-exp2(ORPRT1))U...U Cyc(path-exp2(ORPRTn))

In case 2 of theorem 4.1 we had to prove the relation

Cyc(SEQ1)•Cyc(SEQ2)=Cyc(SEQ1;SEQ2)

Here we have to prove the relation

Cyc(OR1)U Cyc(OR2)=Cyc(OR1,OR2)

where OR1 and OR2 are basic orelements, which may be shown by similar arguments, as those in case 2 of theorem 4.1.
By applying "exp-Cyc2" to a macro orelement we obtain

exp-Cyc2(ORPRT1,...,ORPRTn)=
exp-Cyc2(ORPRT1)U...U exp-Cyc2(ORPRTn))

The two expressions above are the same, provided that for any string produced by "orpart" denoted by ORPRT, the relation

Cyc(path-exp2(ORPRT))=exp-Cyc2(ORPRT)

holds. This relation may be shown by the following cases, depending on whether ORPRT denotes a strict orelement replicator or distributor (cases 6 and 7 respectively), or a strict starelement imbricator (case 8), or a macro starelement (cases 9, 10, 11, 12).

case 6

Applying "path-exp2" to a strict orelement replicator and then "Cyc" we obtain

Cyc(path-exp2(ORREPL))=Cyc(path-exp2(replexp^0(ORREPL)))

and applying "exp-Cyc2" we obtain

exp-Cyc2(ORREPL)=exp-Cyc2(replexp^0(ORREPL))

The equality of the two expressions may be shown by case 5 since

replexp^0(ORREPL)

yields a macro orelement.

case 7

Applying "path-exp2" to a strict orelement distributor and then "Cyc" we obtain

Cyc(path-exp2(ORDISTR))=Cyc(path-exp2(replexp^0(ORDISTR)))

and applying "exp-Cyc2" we obtain

exp-Cyc2(ORDISTR)=exp-Cyc2(replexp^0(ORDISTR))

The equality of the two expressions may be shown by case 5 since

distrexp^0(ORDISTR)

yields a macro orelement.

case 8

Applying "path-exp2" to STARMACRO and then "Cyc" we obtain

Cyc(path-exp2(STARMACRO))=Cyc(path-exp2(replexp^0(STARMACRO)))

and "exp-Cyc2" we obtain

exp-Cyc2(STARMACRO)=exp-Cyc2(replexp^0(STARMACRO)))

The equality of the two expressions is shown by case 9 since

replexp^0(STARMACRO)

yields a starelement.

case 9

Applying "path-exp2" to EL* and then "Cyc" we obtain

Cyc(path-exp2(EL*))=Cyc(path-exp2(EL)*)=Cyc(path-exp2(EL))*

and by applying "exp-Cyc2" we obtain

exp-Cyc2(EL*)=exp-Cyc2(EL)*

The equality of the two expressions depend on the equality of the starred expressions which may be shown by any of the following cases, depending on whether EL denotes an operation (case 10), or an element (case 11), or a strict element imbricator.

case 10

Applying "path-exp2" to OP and then "Cyc" we obtain

Cyc(path-exp2(OP))=Cyc(OP)={OP}

and applying "exp-Cyc2" we obtain

exp-Cyc2(OP)={OP}

yielding the same result.

case 11

Applying "path-exp2" to (MSEQ) and then "Cyc" we obtain

Cyc(path-exp2((MSEQ)))=Cyc((path-exp2(MSEQ)))=Cyc(path-exp2(MSEQ))

and "exp-Cyc2" we obtain

exp-Cyc2((MSEQ))=exp-Cyc(MSEQ)

The equality of the two expressions may be shown by case 2.

case 12

Applying "path-exp2" to ELMACRO and then "Cyc" we obtain

Cyc(path-exp2(ELMACRO))=
Cyc(path-exp2(replexp^0(ELMACRO)))

and applying "exp-Cyc2" we obtain

exp-Cyc2(ELREPL))=exp-Cyc2(replexp^0(ELREPL))

The equality of the above expressions may be shown by case 12, since

$replexp^0$(ELREPL)

yields an element of the form

(MSEQ)

This completes the proof of the theorem.///

The above theorem gives us a shortcut for constructing the cycle set of the expansion of a macro path. Instead of applying two functions "path-exp2" and "Cyc" we may just apply the function "exp-Cyc2" which is of the same order of complexity as "path-exp2".

4.2 CONSTRUCTING ORDERED CYCLE SETS BY EXPANSION OF MACRO-CYCLE OBJECTS

In the previous section we gave rules for constructing the ordered cycle sets of basic programs obtained by the expansion of macro programs, from the macro programs themselves. The ordered cycle sets were constructed in two parts. In the first part all bodyreplicators are expanded and ordered expressions yielding the cycle sets of individual pure macro paths were obtained. In the second part, the cycle sets of individual pure macro paths were obtained by the composition of cycle sets of parts of macro sequences by concatenation or by union operations. This approach yields correct results only when the constituent parts are syntactically strong strings, or, if these involve macro elements, generating syntactically strong strings. This means that by understanding the ordering of operations specified by small parts of a macro sequence, we may understand the ordering of operations specified by the whole path. For the macro paths produced by the grammar of section 3.2 the smallest such parts are the macro orelements. For the macro paths produced by the grammar of section 4.1.2 however, the smallest such parts are the elements or macro elements. Programs produced by the syntax rules of 4.1.2 are more easily readable, in general, than those produced by the syntax of 3.2,

as it is easier to understand a lot of small parts of a macro sequence rather than a few larger ones. Of course, this is achieved at the expense of loss of power of expression, since the syntax of macro elements was restricted. The reading of macro elements produced by the grammar of 4.1.2 is not possible, in general, without them being expanded first, as the regularities they generate may not be syntactically strong and some parts of them may bind with parts of adjacent regularities. For example consider the replicator R1 in P10:

R1 #i:1,3,1[A(i);B(i),@]

which expands to:

A(1);B(1),A(2);B(2),A(3);B(3)

We observe that the operation "B(1)" of the first regularity forms an orelement with the operation "A(2)" of the second regularity and similarly the operation "B(2)" of the second regularity forms an orelement with "A(3)" of the third regularity. The correct reading of R1 is:

do A(1)
followed by B(1) or A(2),
followed by B(2) or A(3),
followed by B(3).

The reading of macro elements is greatly improved when they generate regularities which are syntactically strong strings for two reasons:

1. each regularity may be read independently of the rest, that is no part of any regularity binds with parts of other regularities, and

2. their reading is very similar.

To demonstrate the above points let us consider few replicators generating such regularities and their reading. First the replicator R2:

R2 #i:1,3,1[A(i),B(i);@]

which may be read as:

do A(1) or B(1),
followed by A(2) or B(2),
followed by A(3) or B(3).

Observe that the phrase "A(1) or B(1)" corresponds to the reading of the orelement "A(1),B(1)" in the first regularity which R2 generates, the phrase "A(2) or B(2)" to the reading of "A(2),B(2)", etc.

The replicator R3

R3 #i:1,3,1[(A(i);B(i)),@]

may be read as:

do A(1) followed by B(1),
or A(2) followed by B(2),
or A(3) followed by B(3).

Similarly to R2 above, the phrase "A(1) followed by B(1)" corresponds to the reading of the element "(A(1);B(1))" in the first regularity which R3 generates, the phrase "A(2) followed by B(2)" to the reading of "(A(2);B(2))", etc.

Imbricators are more difficult to read than concatenators and distributors, in general, because the regularities they generate do not follow each other but are nested within each other. However, the reading of imbricators the regularities of which are syntactically strong strings is easier than the reading of the rest. Consider for example the imbricator R4

R4 #i:1,3,1[(SKIP(i);@@),V(i)]

which may be read as:

```
do  SKIP(1)
    followed by     SKIP(2)
                    followed by     SKIP(3)
                    or by     V(3),
    or by     V(2),
or by    V(1).
```

The phrase "SKIP(1)...or V(1)" corresponds to the reading of the outermost regularity "(SKIP(1)...),V(1)", the phrase "SKIP(2)...or V(2)" to the reading of "(SKIP(2)...),V(2)" which follows "SKIP(1)", etc.

The reading of macro elements generating regularities which are syntactically strong strings could be concisely represented. This concise representation is particularly important when the index specification of replicators are parametarized and the number of regularities which macro elements generate is not fixed.

The reading of macro elements is an informal way of describing the ordering of operations they specify. The ease of reading of macro elements generating regularities which are syntactically strong strings may be formally expressed in the construction of the cycle sets of macro paths involving only such macro elements. These cycle sets may be constructed by the composition of the cycle sets of their regularities. For example the cycle set of R2 may be formed by the composition:

$$\{A(1),B(1)\}\circ\{A(2),B(2)\}\circ\{A(3),B(3)\}$$

We may observe that the three string sets in the above expression are very similar and may be obtained by replacing "i" indexing the operations in the string set

$$\{A(i),B(i)\}$$

by the values 1,2,3, the values in the range of the replicator index. The above set may be considered as the cycle set of the general regularity inside "[ ]" of R2, ignoring the ";@". If all replicators and distributors in sequences had this property then they would not in

principle have to be expanded in order to find their cycle set. The cycle set of their general regularity would be sufficient to generate the cycle set of the whole replicator or distributor. Furthermore, since macro elements would not have to be expanded, the bodyreplicators would not have to be expanded as well. We cannot in general avoid the expansion of bodyreplicators in the approach in the previous section, since the range of indices of replicators in macro sequences may depend on bodyreplicator indices, implying that for the replicators and distributors in macro sequences to be expanded the bodyreplicators have to have been expanded first. This leads to the idea of macro cycle objects constructed from macro programs which represent ordered cycle sets of basic programs as economically as macro programs represent basic programs, and from which ordered cycle sets may be generated in the same way as basic programs are generated from macro programs.

These macro cycle objects besides being a formal means for representing the ordered cycle sets of an expanded macro program, they also aid the verification of macro programs. Strictly speaking, all verification methods and techniques developed in COSY apply to basic programs only. This has the disadvantage that a macro program cannot be verified, unless it is expanded first, implying that all its parameters have to be given specific values. The consequence of this is that macro programs cannot be verified for all values of their parameters. This limitation was overcome by adopting informal techniques, as in [SL78], which made possible the verification of parametarized macro programs. When verifying a COSY program, we frequently argue in terms of the firing sequences of paths, which are constructed by their cycle sets (cf. section 2.2). Thus, we are confronted with the task of representing the firing sequences of the macro paths of macro programs. As these paths may involve macro elements generating a finite but indefinite number of regularities the representation of the general cycle sets is fundamental.

An informal approach for representing repetition of patterns in the elements of the cycle sets was followed in [SL78] using ellipses. For example the cycle set of the path involving the replicator R4

path #i:1,m,1[(SKIP(i);@@),V(i)] end

was represented by:

```
{V(1),
 SKIP(1).V(2),
 SKIP(1).SKIP(2).V(3),
      ...
 SKIP(1). ... .SKIP(m-1).V(m),
 SKIP(1). ... .SKIP(m)}
```

The ellipses in the above cycle set denote two kinds of repetition patterns. The ellipses denote repetition of operations SKIPs of the form:

skip_rep(j)=SKIP(1). ... .SKIP(j) for $1 \leq j \leq m$

but also denote repetition of cycles of the form:

skip_rep(1).V(2),skip_rep(2).V(3), ... ,skip_rep(m-1).V(m)

Even expressing the cycles of this relative simple path by ellipses is cumbersome. As macro elements may in general, be nested inside other macro elements the precise representation of cycle sets using ellipses becomes an impossible task. We need a notation for the concise representation of ordered cycle sets of macro programs from which the ordered cycle sets of the expanded basic program could be generated by expansion. This notation should be able to represent sets of cycles or cycles of all macro elements be it bodyreplicators, concatenators, distributors or imbricators. For this representation to be possible though, all macro elements in macro sequences should always generate regularities which are syntactically strong strings.

In the next subsection 4.2.1 we constrain some of the syntax rules of 4.1.2 to produce macro programs the macro paths of which involve concatenators, distributors and imbricators the regularities of which are syntactically strong strings and we define the function "expand2" by which these programs are expanded. In subsection 4.2.2 we define a notation for concisely representing cycle sets of macro programs produced by this grammar, we define the function "m-Cycs" for obtaining

macro cycle objects in this notation from macro programs and define the function "exp-Cyc" by which these objects are expanded yielding ordered cycle sets. Finally, in subsection 4.2.3 we prove that the ordered cycle sets of the expansion of a macro program produced by the grammar of 4.2.1 are the same as the ordered cycle sets we obtain from the expansion of the macro cycle objects of the macro program.

4.2.1 Syntax and Expansion Rules of Constrained macro-Programs

In the grammar of this section we constrain the production rules in section 4.1.2, or, to be more precise, those in appendix C, in order to produce concatenators, distributors and imbricators the expansion of which and each of their regularities are syntactically strong strings. This is achieved by forcing the main connective of the string generated by expansion to separate each regularity. Actually, the only macro elements the syntax of which needs to be constrained are those which generate sequences since the main connective of their expansion, namely ";", does not always separate the regularities. The regularities in the expansion of the rest of the macro elements are orelements, starelements and elements and consequently are syntactically strong strings in any of their contexts. Therefore, we only need to constrain the production rules for "seqconcseq", "seqimbrseq" and "seqdistr". The non-terminals "seqdistr" and "seqconcseq", producing distributors generating sequences and strings inside "[ ]" of a sequence concatenator, respectively, will be redefined by:

seqdistr=;{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]

seqconcseq={seqpart;}+@

The difference with corresponding rules rules of section 4.1.2 for "seqconcseq" and "seqdistr" is that here we eliminated the production of "," as the connective separating regularities generated by concatenators and distributors.

We have yet to constrain the imbricators generating sequences produced by the syntax in 4.1.2 in which each regularity is a syntactically strong string. An imbricator

```
#i:1,n,1[p(i) @ t @ q(i)]
```

generates three kinds of regularities in general:

1. p(i)...q(i) when $1 \leq i \leq n-1$,
2. p'(n) t q'(n) when $i=n$
3. t' when $n<1$

as we may recall from sections 3.2 and 3.3.

For imbricators generating sequences each regularity they generate is a sequence and for it to be syntactically strong it should be between any of "(", ";" and any of ")", ";". These regularities appear in the same context as that of "@ t @" and consequently, the string "@ t @" should be between any of "(", ";", "[" and any of ")", ";", "]". The two extra terminal symbols "[" and "]" in the context of "@ t @" arise from the fact that when a replicator expands these disappear and the context of the expanded string is the context of the imbricator itself. The context of the imbricator generating sequences is any of "(", ";", "path" on its left and any of ")", ";", "end" on its right which guarantee that the outermost regularity and consequently the whole expansion of an imbricator is syntactically strong.

The syntax rules for "seqimbrseq" producing strings inside "[ ]" of a sequence replicator then should be:

```
seqimbrseq=seqimbr_at_seq
          /{seqpart;}+ seqimbror {; seqpart}*
          /{seqpart;}* seqimbror {; seqpart}+

seqimbror={orpart,}* seqimbrstarel {, orpart}*

seqimbrstarel=seqimbrel/seqimbrel*

seqimbrel=(seqimbrseq)

seqimbr_at_seq=
       {seqpart;}+ {@/at_orlf} {;seqpart}* ; {@/at_orlb} {;seqpart}+
      /{seqpart ;}+ at_or2fb {; seqpart}+
      /@{seqpart ;}* {@/at_orlb} {; seqpart}+
      /{seqpart ;}+ {@/at_orlf} {; seqpart}*@
      /@ msequence @
```

For the imbricators generating orelements, starelements and elements each of their regularities must be orelements, starelements and elements respectively. These regularities are syntactically strong in any context the string "@ t @" is in. The syntax rules for them will still be those of section 4.1.2 or more accurately those of appendix C. The complete syntax for programs involving only these macro elements may be found in appendix D. Syntax rules in appendix D are associated with mnemonic names starting with "MN" to denote syntax rules developed in section 3.2, or with "RN" to denote syntax rules developed in section 4.1.2 (appendix C), or with "CN" to denote syntax rules developed in this section.

Let us now give some examples of imbricators the regularities of which are syntactically strong in their expansion. The imbricator is R5 produced by "seqmacro"

```
R5 #i:1,3,1[(A(i);@@),B(i);C(i)]
```

and expands to the sequence E(R5)

```
E(R5) (A(1);(A(2);(A(3)),B(3);C(3)),B(2);C(2)),B(1);C(1)
```

The imbricator R6 is produced by "ormacro"

```
R6 #i:1,3,1[(SKIP(i);@@),(CS_BEGIN(i);CS_END(i))]
```

and expands to the orelement E(R6)

```
E(R6) ( SKIP(1)
       ; ( SKIP(2)
          ; (SKIP(3))
           ,(CS_BEGIN(3);CS_END(3))
         )
        ,(CS_BEGIN(2);CS_BEGIN(2))
      )
     ,(CS_BEGIN(1);CS_END(1))
```

The imbricator R7 is produced by "starmacro"

```
R7 #i:1,3,1[(UP(i);@;full*;@;DOWN(i))*]
```

and expands to the starelement

```
E(R7)( UP(1)
      ;( UP(2)
        ;( UP(3)
          ;full*
          ;DOWN(3)
         )*
        ;DOWN(2)
       )*
      ;DOWN(1)
     )*
```

Finally, the imbricator R8 is produced by "elmacro"

```
R8 #i:1,3,1[(C(i);R(i),@@)]
```

and expands to the element

```
E(R8)( C(1)
     ; R(1)
      ,( C(2)
       ; R(2)
        ,(C(3);R(3))
       )
     )
```

The reading of macro elements generating regularities which are syntactically strong strings could be concisely represented. This is particularly important when the index specification of replicators and the sizes of the arrays are parametarized and as a result the number of regularities to be generated is not fixed. Concatenators and distributors generating sequences of the form

```
#i:1,n,1[p(i) ;@]
;[p]
```

respectively, may be read as:

```
for all i=1,...,n do consecutively p(i)
```

Concatenators and distributors generating orelements of the form

```
#i:1,n,1[p(i) ,@]
,[p]
```

respectively, may be read as

```
for any i=1,...,n do p(i)
```

For example, R2 and R3 when the final value of their indices is parametarized by the integer n may be read as:

```
for all i=1,...,n do consecutively A(i) or B(i)
```

and

```
for any i=1,...,n do A(i) followed by B(i)
```

respectively. Imbricators are more difficult to read than concatenators and distributors as regularities are nested and not following each other. Imbricators generating syntactically strong strings

#i:1,n,1[p(i) @ t @ q(i)]

may be read recursively by defining the reading of their general regularity as follows:

read_reg(i)= if i<n do p(i) read_reg(i+1) q(i)
if i=n do p'(n) t q'(n)
if i>n do t'

Then the general reading of imbricators is given by

read_reg(1)

When the final value of the indices of imbricators R5, R6, R7, R8 is parametarized by the integer n, then the imbricators may be read by

read_reg(1)

where read_reg(i) for R5 is

read_reg(i)= if i<n do A(i) followed by read_reg(i+1) or B(i),
followed by C(i)
if i=n do A(n) or B(n), followed by C(n)

For i>n the expansion of R5 is empty and consequently R5 is not valid. The reading of the general regularity of R6 is:

read_reg(i)= if i<n do SKIP(i) followed by read_reg(i+1),
or CS_BEGIN(i) followed by CS_END(i)
if i=n do SKIP(i)
or CS_BEGIN(n) followed by CS_END(n)

For i>n the expansion of R6 is empty and consequently R6 is not valid. The reading of the general regularity of R7 is

```
read_reg(i)= if i<n do repeat:UP(i)
                                followed by read_reg(i+1)
                                followed by DOWN(i)
             if i=n do repeat:UP(n)
                                followed by repeat:full,
                                followed by DOWN(n)
             if i>n do repeat:full
```

The general regularity of R8 may be read as

```
read_reg(i)= if i<n do C(i), followed by R(i) or by read_reg(i+1)
             if i=n do C(n) followed by R(n)
```

Any program produced by the syntax rules of appendix D may also be produced by the syntax rules of the section 4.1.2 (*) and as we have constrained and not extended the syntax rules generate basic programs when expanded. Similarly to the strict sequence macro elements of section 4.1.2, the macro elements produced by the above syntax rules may also generate orelements, starelements and elements, when their index range consists of one value or it is empty.

Here we will define the expansion of macro programs in an alternative way from that of sections 3.3 and 4.1.2 by defining a function "expand2". The necessity for an alternative expansion is of a technical nature. As we like to find the cycles of the general regularity inside "[ ]" of a macro element, we cannot consider this regularity as a string. We need to decompose regularities into their syntactic entities on which a function "m-Cycs" will apply yielding ultimately the macro cycles of these regularities. As both "expand2" and "m-Cycs" apply to macro programs, it would be convenient, in showing that the ordered cycle sets of the expansion of macro programs are the same as the expansion of the macro cycle objects of macro programs, if both functions applied to the same syntactic entities of macro programs. Since the functions "expand", "replexp^0" and "distrexp^0" of 3.3 treat regularities as strings and do not decompose them into their syntactic entities, an alternative definition for the expansion of macro programs will be given, in terms of the function "expand2".

(*) production rules 4.1.2 are subset of production rules in 4.1.3

In the definition of function "expand2" we do not make use of any of the auxilliary functions "replexp^0", "distrexp^0", we used in "expand" of section 3.3.3 and "expand1" of section 4.1.2. In "expand2" the expansion of macro elements is defined in an alternative way. We have distinguished two kinds of macro elements: bodyreplicators and concatenators the expansion of which is defined by iteration and imbricators the expansion of which is obtained by recursion. The expansion of distributors is obtained from the expansion of their equivalent replicators (cf. section 3.3.2). We have made this distinction between bodyreplicators and concatenators on one hand and imbricators on the other, since the former generate strings which would be produced by iterative productions of a non-terminal, whilst the latter generates strings which would only be produced, in general, by recursive productions of non-terminals, as regularities are nested within each other. The expansion of a bodyreplicator, for example will be defined by

expand2(#i:1,n,1[PBRs(i)])=expand2(PBRs(1))...expand2(PBRs(n))

The expansion of imbricators will be defined recursively, generating at each level of the recursion one regularity. The regularities are obtained by substituting in "p(i) @t@ q(i)" the appropriate value for "i". These strings are considered to be special macro "sequences", involving "@t@" as a special non-starred "element". The expansion of these macro "sequences" will be defined, similarly to the expansions of proper macro sequences, by expanding its syntactic sub-entities. As the syntactic entities of this macro sequence are expanded, the expansion of the element "@t@" will be eventually needed. We consider the expansion of this special element to be the next regularity of the imbricator. The problem is that when we reach that point we do not know the imbricator the string "@t@" corresponds to. One solution would be to pass together with each syntactic entity of the regularity the imbricator as a second argument to "expand2". But this would mean that for some syntactic entities "expand2" would be a one argument function and for others a two argument function. For this reason we decided on another solution. We assume the existence of a stack in which a copy of the imbricator is to be saved whilst syntactic entities of its regularities are expanded. This copy will be needed when the element

"@ t @" is "expanded". The reason we use a stack is that the regularity of an imbricator could involve other imbricators, each having its own "@ t @" string, which "expand" to different strings. We associate two operations with this stack: "imbr-push" by which imbricators are pushed into the stack and "imbr-pop" by which they are poped out of the stack.

We may use this stack as follows: When an imbricator is to generate at least one of its general regularities, determined by its index specification, the imbricator is pushed into the stack with the lower value of its index incremented by one. The expansion of this stacked imbricator will generate all the inner regularities to the current regularity of the original imbricator. After the modified imbricator is stacked its current regularity may be expanded. As the expansion of syntactic entities of this regularity are expanded, the expansion of the special element "@ t @" will be eventually needed. Its expansion is defined to be the expansion of the imbricator at the top of the stack. When the imbricator generates its last regularity it will not be stacked and as this last regularity will not contain the special element "@ t @" the expansion of the original imbricator will terminate correctly.

In the definition of "expand2" which follows the syntactic variables MPBODY, CPBRi for i=1,...,n, COLs, PBRi for i=1,...,n and PBRs denote the same syntactic entities as defined in section 4.1. Paths will be represented by:

path MSEQ end

where MSEQ denotes a macro sequence which is represented by

SEQPRT1;...;SEQPRTn

where each of SEQPRTi for i=1,...,n denotes a macro element produced by "seqmacro" or a macro orelement. A macro orelement is represented by

ORPRT1,...,ORPRTn

where each of ORPRTi for i=1,...,n denotes either a macro element produced by "orpart" or "starmacro" or a starred element. A starelement

is represented by

EL* or EL

where EL denotes a macro element produced by "elmacro", or an operation represented by

OP

or an element of the form (msequence) represented by

(MSEQ)

Concatenators and distributors produced by "seqmacro" will be represented by

#i:1,n,1[MSEQ(i);@]
;[MSEQ]

respectively, where MSEQ(i) denotes a macro sequence some operations of which may depend on "i", and MSEQ denotes a macro sequence involving array slices instead of operations.

Concatenators and distributors produced by "ormacro" will be represented by

#i:1,n,1[MOR(i),@]
,[MOR]

respectively, where MOR(i) denotes a macro orelement some operations of which may depend on "i" and MOR denotes a macro orelement involving array slices instead of operations.

All imbricators produced by "seqmacro", "ormacro", "starmacro" and "elmacro" will be represented by

#i:1,n,1[p(i) @t@ q(i)]

The string "p(i) @t@ q(i)" in the above representation of an imbricator may be of four different forms each corresponding to one of the four types of imbricators. In order to keep the definitions of the functions "expand2" and "m-Cycs" as short as possible we shall not distinguish similar syntactic entities of the four forms. In the formal grammar of appendix D we had to have four groups for syntax rules producing imbricators for two reasons:

1. to specify that the string "t" between the two "@"s is different in each case, and

2. to specify that the context of "@t@" in imbricators generating sequences excludes commas.

If we regard the string "@ t @" as one entity, the first reason for their distinction is not important. As for the second reason we may for the moment ignore it when defining syntactic entities. All syntactic variables except the syntactic variable representing the string "@t@" will be sufficed by "(i)" to denote that integer expressions in the strings they represent, may depend on "i", denoting the index of replicators.

We may represent the string "p(i) @t@ q(i)" of a genuine imbricator by

IM_SP1(i);...;IM_SPn(i)

where IM_SPj(i) for j=1,...,n denote strings produced by "seqpart", except exactly one which involves the string "@t@" corresponding to the imbricator, which may be represented by the orelement

IM_OP1(i),...,IM_OPn(i)

If an imbricator generates an orelement, a starelement or an element, the each of IM_OPj(i) for j=1,...,n denotes a string produced by "orpart", except exactly one which involves the string "@t@" corresponding to this imbricator. The entity which involves "@t@" is the whole of the string "p(i) @t@ q(i)" of imbricators generating

starelements and elements. This entity entity may be represented by

IM_EL(i)* or IM_EL(i)

The syntactic variable IM_EL(i) denotes an element involving "@ t @" and may be represented by

(IMBR_SEQ(i))

in which IMBR_SEQ(i) denotes a special sequence which involves "@t@" and may be represented by either

IM_SP1(i);...;IM_SPn(i)

if "@t@" is further nested inside "( )", or by

AT_SP1(i);...;AT_SPn(i)

when "@t@" is not further nested inside "( )". Each of AT_SPj(i) for j=1,...,n denotes a string produced by "seqpart", except exactly one which in the case of imbricators generating sequences is "@t@", represented by

AT_EL

but in the case of imbricators generating orelements, starelements and elements is an orelement involving "@t@" as an element and may be represented by

AT_OP1(i),...,AT_OPn(i)

where each of AT_OPj(i) for j=1,...,n denotes strings produced by "orpart" except exactly one which is the special element "@t@" which we have represented by AT_EL.

Finally, the string "p(i) @t@ q(i)" of non-genuine imbricators generating sequences may be represented by

AT_SP1(i);....;AT_SPn(i)

defined as above. The string "p(i) @t@ q(i)" of non-genuine imbricators generating orelements may be represented by

AT_OP1(i),...,AT_OPn(i)

as explained above.

Let us now define the function "expand2":

```
expand2(e)=cases e:

1.  program MPBODY endprogram      -> program expand2(MPBODY) endprogram
2.  CPBR1...CPBRn                  -> expand2(CPBR1)...expand2(CPBRn)
3.  COLs PBR                       -> expand2(PBR)
4.  #i:1,m,1[PBRs(i)]              -> expand2( PBRs(1))...expand2(PBRs(m))
5.  PBR1...PBRn                    -> expand2(PBR1)...expand2(PBRn)
6.  path MSEQ end                  -> path expand2(MSEQ) end
7.  SEQPRT1;...;SEQPRTn            -> expand2(SEQPRT1);...;expand2(SEQPRTn)
8.  #i:1,n,1[MSEQ(i);@]            -> expand2(MSEQ(1));...;expand2(MSEQ(n))
9.  ;[MSEQ]                        -> expand2(MSEQ(1));...;expand2(MSEQ(n))
10. ORPRT1,...,ORPRTn              -> expand2(ORPRT1),...,expand2(ORPRTn)
11. #i:1,n,1[MOR(i),@]             -> expand2(MOR(1)),...,expand2(MOR(n))
12. ,[MOR]                         -> expand2(MOR(1)),...,expand2(MOR(n))

13. #i:in,n,1[p(i)@ t @q(i)]       ->  if in<n then
                                           imbr-push(#i:in+1,n,1[p(i)@t@q(i)])
                                           expand2(p(in)@ t @q(in))
                                       if in=n then expand2(p'(n) t q'(n))
                                       if in>n then expand2(t')

14. IM_SP1(k);...;IM_SPn(k)        -> expand2(IM_SP1(k));...;expand2(IM_SPn(k))
15. IM_OP1(k),...,IM_OPn(k)        -> expand2(IM_OP1(k)),...,expand2(IM_OPn(k))
16. IM_EL(k)*                      -> expand2(IM_EL(k))*
17. (IMBR_SEQ(k))                  -> (expand2(IMBR_SEQ(k))
18. AT_SP1(k);...;AT_SPn(k)        -> expand2(AT_SP1(k));...;expand2(AT_SPn(k))
19. AT_OP1(k),...,AT_OPn(k)        -> expand2(AT_OP1(K)),...,expand2(AT_OPn(k))
20. AT_EL                          -> expand2(imbr-pop)
21. EL*                            -> expand2(EL)*
22. OP                             -> OP
23. (MSEQ)                         -> (expand2(MSEQ))
```

We will not formally prove that the expansion of a macro program MPROG, given by expand2(MPROG), is a basic program as it may be proven in a style we have proven theorem 3.11 of section 3.3.3.

We have to point out a subtle operational difference between the functions "expand2" and the rest of the functions defined by cases, such as "expand", "expand1", "Cycles", "exp-Cycls", etc., due to the use of the stack in "expand2". When a function applies to syntactic variables which themselves have as components other syntactic variables, the result is defined in terms of the partial results obtained by applying the same function on these components. The order of evaluation of these partial results is not important. This is true for all our functions defined by cases. In evaluating partial results of "expand2", however, some fixed order should be followed, whilst in other functions these evaluations could be performed concurrently. The reason for this difference is that "expand2" uses a common stack which should be accessed orderly. For otherwise, if imbricators are pushed and poped unorderly, an imbricator may be expanded at a wrong position.

4.2.2 Macro Cycle Objects and their Expansion

Let us now examine what kind of features we need to represent sets of cycles of macro programs concisely, and suggest a reasonable notation incorporating these features.

The macro cycles of a macro program body will be wrapped between the word pair "mcycles" and "endmcycles" and the macro cycles of a macro path between "pcyc" and "endpcyc". The macro cycle sets of bodyreplicators and paths will be separated by "&", macro cycle sets of strings produced by "seqpart" will be separated by "⊙", and macro cycle sets of strings produced by "orpart" will be separated by "U". As "," has precedence over ";" in macro sequences so "U" will have precedence over "⊙" in macro cycle sets.

We need four other features in this notation:

1. one to represent the union of similar sets, for representing the

union of cycle sets of the regularities of concatenators and distributors generating orelements,

2. another to represent the concatenation of similar cycle sets, for representing the concatenation of cycle sets of the regularities of concatenators and distributors generating sequences,

3. a third one to represent the imbrication of similar cycle sets, for representing the cycle sets of the regularities of imbricators, and finally

4. one to represent the ordering of similar cycle sets, for representing ordered cycle sets of paths in the regularities of bodyreplicators.

As we have already used the symbol "U" for the set union operator, it is natural to use the notation

$$\bigcup_{i=1}^{n}[S(i)]$$

to represent the union of the sets S(1),...,S(n)

$$S(1) \cup \ldots \cup S(n)$$

where S(i) denotes a string set expression, involving sets of a single operation name and macro sets, the integer expressions in which may depend on "i", and S(j) for j=1,...,n is obtained from S(i) by replacing the index "i" by one of the values for j. For example the expression

$$\bigcup_{i=1}^{3}[\{DEPOSIT(i)\}]$$

represents the union of sets

$$\{DEPOSIT(1)\} \cup \{DEPOSIT(2)\} \cup \{DEPOSIT(3)\}$$

As we have used the symbol "$\circ$" for the concatenation of sets of strings, we shall use the notation

$$\overset{n}{\underset{i=1}{\circ}}[S(i)]$$

to represent concatenation of the sets S(1),...,S(n)

$$S(1) \circ ... \circ S(n)$$

where S(i) and S(j) for j=1,...,n are defined as above. For example, the expression

$$\overset{3}{\underset{i=1}{\circ}}[\{DEPOSIT(i)\}]$$

represents the concatenation of sets

$$\{DEPOSIT(1)\}\circ\{DEPOSIT(2)\}\circ\{DEPOSIT(3)\}$$

Similarly, we shall use the notation

$$\overset{n}{\underset{i=1}{\&}}[S(i)]$$

to represent the ordering of collections of cycle sets S(1),...,S(n)

$$S(1) \& ... \& S(n)$$

where S(i) and S(j) for j=1,...,n are defined as above.

For example the expression

$$\overset{3}{\underset{i=1}{\&}}[\underline{\text{pcyc}}\ \{DEPOSIT(i)\}\circ\{REMOVE(i)\}\ \underline{\text{endpcyc}}]$$

represents the ordering of the sets of cycles

$$\underline{\text{pcyc}}\ \{DEPOSIT(1)\}\circ\{REMOVE(1)\}\ \underline{\text{endpcyc}}\ \&$$

pcyc {DEPOSIT(2)}•{REMOVE(2)} endpcyc &
pcyc {DEPOSIT(3)}•{REMOVE(3)} endpcyc

The macro cycle sets of the paths

path #i:1,3,1[DEPOSIT(i),@] end
path #i:1,3,1[DEPOSIT(i);@] end

and of the the ordering of the cycle sets of the paths generated by the bodyreplicator

#i:1,3,1[path DEPOSIT(i);REMOVE(i) end]

may be concisely represented by

pcyc $\overset{3}{\underset{i=1}{U}}$[{DEPOSIT(i)}] endpcyc

pcyc $\overset{3}{\underset{i=1}{\bullet}}$[{DEPOSIT(i)}] endpcyc

$\overset{3}{\underset{i=1}{\&}}$[pcyc {DEPOSIT(i)}•{REMOVE(i)} endpcyc]

respectively.

Let us now examine what kind of a notation we need to represent concisely the cycle set of an imbricator. We may recall that an imbricator

#i:1,n,1[p(i)@ t @q(i)]

generates either the string Exp1

Exp1. t'

when 1>n, or the string Exp2

Exp2. p(1) p(2)...p'(n) t q'(n)...q(2) q(1)

when $n \geq 1$. The string Exp2 involves two kinds of regularities in general:

p(i)...q(i) for i=1,...,n-1

and

p'(n) t q'(n)

Therefore for the concise representation of the cycles of the regularity of any imbricator we need in general the macro cycles of

1. t'

2. p'(n) t q'(n)

3. p(i)...q(i) for i=1,...,n-1

Of the three of the above expressions 1 and 2 are not repeated in the imbricator expansion and must therefore, be considered individually; the regularity which is repeated in the expansion of an imbricator is of the form 3. This leads us to adopt the following notation for representing the cycles of the regularities of imbricators:

$$\overset{n}{\underset{i=1}{\uparrow}}[A(i)/B/C]$$

where A(i), B, C denote the macro cycle expressions of "p(i)@ t @q(i)", "p'(n) t q'(n)" and "t'" respectively.

As the regularity of the form 3 will always imbricate other regularities, we must indicate in A(i) where the cycle set of the inner regularities are to appear. We do that by using the symbol "↓". As the inner regularity is to appear in the context of "@ t @" we shall regard the cycle set of this special element "@ t @" as being "↓".

For example the macro cycle expression

$$\uparrow_{i=1}^{3}[(\{C(i)\}\circ\{R(i)\}U\downarrow)/(\{C(3)\}\circ\{R(3)\})/\{\lambda\}]$$

where "λ" denotes the empty string, represents the set-expression

$$(\{C(1)\}\circ\{R(1)\}U(\{C(2)\}\circ\{R(2)\}U(\{C(3)\}\circ\{R(3)\})))$$

which is the cycle set of the sequence

$$(C(1);R(1),(C(2);R(2),(C(3);R(3))))$$

which may be obtained by the expansion of the imbricator R8

$$R8\ \#i{:}1,3,1[(C(i);R(i),@@)]$$

The macro cycle set representing the cycles of the string obtained by the expansion of replicator R4 is

$$\uparrow_{i=1}^{n}[(\{SKIP(i)\}\circ\downarrow)\ U\ \{V(i)\}/(\{SKIP(n)\})\ U\ \{V(n)\}/\{\lambda\}]$$

The macro cycle object of a macro program may be constructed formally by the function "m-Cycs" defined below, which applies to the same syntactic entities as function "expand2".

m-Cycs(e)=cases e:

1. program MPBODY endprogram → mcycles m-Cycs(MPBODY) endmcycles
2. CPBR1...CPBRn → m-Cycs(CPBR1)...m-Cycs(CPBRn)
3. COLs PBR → m-Cycs(PBR)
4. #i:1,n,1[PBRs(i)] → $\overset{n}{\underset{i=1}{\&}}$[m-Cycs(PBRs(i))]
5. PBR1...PBRn → m-Cycs(PBR1)&...&m-Cycs(PBRn)
6. path MSEQ end → pcyc m-Cycs(MSEQ)endpcyc
7. SEQPRT1;...;SEQPRTn → m-Cycs(SEQPRT1)⊙...⊙ m-Cycs(SEQPRTn)
8. #i:1,n,1[MSEQ(i);@] → $\overset{n}{\underset{i=1}{⊙}}$[m-Cycs(MSEQ(i))]
9. ;[MSEQ] → $\overset{n}{\underset{i=1}{⊙}}$[m-Cycs(MSEQ(i))]
10. ORPRT1,...,ORPRTn → m-Cycs(ORPRT1)U...U m-Cycs(ORPRTn)
11. #i:1,n,1[MOR(i),@] → $\overset{n}{\underset{i=1}{U}}$[m-Cycs(MOR(i))]
12. ,[MOR] → $\overset{n}{\underset{i=1}{U}}$[m-Cycs(MOR(i))]
13. #i:in,n,1[p(i) @ t @ q(i)] → $\overset{n}{\underset{i=in}{↑}}$[m-Cycs(p(i) @ t @ q(i))/
    m-Cycs(p'(n) t q'(n))/m-Cycs(t')]
14. IM_SP1(i);...;IM_SPn(i) → m-Cycs(IM_SP1(i))⊙...⊙ m-Cycs(IM_SPn(i))
15. IM_OP1(i),...,IM_OPn(i) → m-Cycs(IM_OP1(i))U...U m-Cycs(IM_OPn(i))
16. IM_EL(i)* → m-Cycs(IM_EL(i))*
17. (IMBR_SEQ(i)) → (m-Cycs(IMBR_SEQ(i))
18. AT_SP1(i);...;AT_SPn(i) → m-Cycs(AT_SP1(i))⊙...⊙ m-Cycs(AT_SPn(i))
19. AT_OP1(i),...,AT_OPn(i) → m-Cycs(AT_OP1(i))U...U m-Cycs(AT_OPn(i))
20. AT_EL → ↓
21. EL* → m-Cycs(EL)*
22. OP → {OP}
23. (MSEQ) → (m-Cycs(MSEQ))

The macro cycle object of the ring buffer with one producer and one consumer specified by the following macro program MPROG1

```
MPROG1

program
  array DEPOSIT REMOVE(n) endarray
  #i:1,n,1[path DEPOSIT(i);REMOVE(i) end]
  path ;[DEPOSIT] end
  path ;[REMOVE]  end
  path ,[DEPOSIT] end
  path ,[REMOVE]  end
endprogram
```

obtained by the function "m-Cycs" is:

```
mcycles

  n
  &[pcyc {DEPOSIT(i)}o{REMOVE(i)} endpcyc] &
 i=1

  pcyc  n
        o[{DEPOSIT(i)}] endpcyc &
       i=1

  pcyc  n
        o[{REMOVE(i)}] endpcyc &
       i=1

  pcyc  n
        U[{DEPOSIT(i)}] endpcyc &
       i=1

  pcyc  n
        U[{REMOVE(i)}] endpcyc
       i=1

endmcycles
```

The macro cycle object of the priority resource manager [LT78, LS78] specified by the macro program MPROG2

:MPROG2

```
program
  array
    DEPOSIT REMOVE(n,m)
    SKIP CS_BEGIN CS_END(m)
  endarray
  #j:1,m,1
   [path ;[DEPOSIT(,j)] end
    path ;[REMOVE(,j)] end
    #i:1,n,1[path DEPOSIT(i,j);REMOVE(i,j) end]
    path SKIP(j),(,[REMOVE(,j)];CS_BEGIN(j);CS_END(j)) end
   ]
  path #j:1,m,1[(SKIP(j);@@),(CS_BEGIN(j);CS_END(j))] end
endprogram
```

obtained by "m-Cycs" is

```
mcycles

   m                n
  &[ pcyc          o[{DEPOSIT(i,j)}] endpcyc &
  j=1              i=1

      pcyc         n
                   o[{REMOVE(i,j)] endpcyc &
                   i=1

       n
      &[{DEPOSIT(i,j)}o{REMOVE(i,j)}] &
      i=1

      pcyc
        {SKIP(j)}U(U[{REMOVE(i,j)}]o{CS_BEGIN(j)}o{CS_END(j)})
                   i=1
      endpcyc

    ] &

  pcyc

     m
    *[({SKIP(j)}o+)U({CS_BEGIN(j)}o{CS_END(j)})/
    j=1
        ({SKIP(n)})U({CS_BEGIN(n)}o{CS_END(n)})/{λ}]

  endpcyc

endmcycles
```

Let us now give general concise readings for the macro cycle sets. Macro cycle sets of the form

$$\overset{n}{\underset{i=1}{\bullet}}[S(i)]$$

may be read as:

for all i=1,...,n do consecutively S(i).

For example the macro set

$$\overset{n}{\underset{i=1}{\bullet}}[\{A(i)\}U\{B(i)\}]$$

may be read as

for all i=1,...,n do consecutively A(i) or B(i).

Macro sets of the form

$$\overset{n}{\underset{i=1}{U}}[S(i)]$$

may be read as

for any i=1,...,n do S(i).

For example the macro set

$$\overset{n}{\underset{i=1}{U}}[(\{A(i)\}\bullet\{B(i)\})]$$

may be read as

for any i=1,...,n do A(i) followed by B(i).

Finally macro sets of the form

$$\overset{n}{\underset{i=1}{\uparrow}}[S(i)/U/T]$$

may be read recursively by defining the reading of the i'th regularity as follows:

```
read_cycreg(i)= if i<n then S(i)
                if i=n then U
                if i>n then T
```

where the symbol "↑" must appear in S(i) standing for "read_cycreg(i+1)". For example the cycle set

$$\overset{n}{\underset{i=1}{\uparrow}}[(\{SKIP(i)\}\circ\uparrow)\cup\{V(i)\}/(\{SKIP(n)\}\cup\{V(n)\}/\{\lambda\}]$$

may be read as:

```
read_cycreg(1)
```

where the reading of the i'th regularity read_cycreg(i) is

```
read_cycreg(i)= if i<n do SKIP(i) followed by read_cycreg(i+1),
                          or V(i)
                if i=n do SKIP(n) or V(n)
```

For programs involving simple macro elements there is no real practical advantage in reading macro cycle objects of macro programs rather than reading the programs themselves. However, for programs involving more complicated macro elements, macro cycle objects have an advantage and are useful in that aspect as well. In certain cases we may simplify the macro cycle expressions by the composition rules of "∘" and "∪" of sets of strings and by applying some relations regarding the union and concatenation operations on sets of strings and macro cycle sets, such as:

1. $A \circ B \cup C = (A \circ B) \cup (A \circ C)$
   $B \cup C \circ A = (B \circ A) \cup (C \circ A)$
2. $A \cup B = B \cup A$
3. $A \cup (B \cup C) = (A \cup B) \cup C = A \cup B \cup C$
4. $A \circ (B \circ C) = (A \circ B) \circ C = A \circ B \circ C$
5. $(A \cup B) \circ C = A \cup B \circ C$

where A, B and C are sets of strings, or macro cycle sets. In addition when A is a string set or a macro set representing union of similar sets, the following property holds:

6. $(A) = A$

Finally when A is a set of strings the following properties hold:

7. $\overset{n}{\underset{i=1}{\cup}}[S(i)] \circ A = \overset{n}{\underset{i=1}{\cup}}[S(i) \circ A]$

8. $A \circ \overset{n}{\underset{i=1}{\cup}}[S(i)] = \overset{n}{\underset{i=1}{\cup}}[A \circ S(i)]$

Consider for example the macro cycle of the replicator R9

R9 #i:1,n,1[((A(i);B(i)),C(i);D(i)),@]

in which none of the parentheses in the regularity are redundant. By applying the function "m-Cycs" we obtain the macro cycle set:

$$\overset{n}{\underset{i=1}{\cup}}[((\{A(i)\} \circ \{B(i)\}) \cup \{C(i)\} \circ \{D(i)\})]$$

which is a quite complicated expression and certainly not more readable than the replicator itself. The above macro set may be read as:

for any i=1,...,n do
    A(i) followed by B(i), or C(i)
    followed by D(i).

We may apply some of the above well defined properties to simplify this

macro set and its reading. The cycle set of the regularity of the above set expression is equivalent to

(({A(i).B(i)}) U {C(i)}•{D(i)})=	by composition of "•"
({A(i).B(i)} U {C(i)}•{D(i)})=	by rule 6
({A(i).B(i),C(i)}•{D(i)})=	by composition of "U"
({A(i).B(i).D(i),C(i).D(i)})	by composition of "•"
{A(i).B(i).D(i),C(i).D(i)}	by rule 6

which means that the macro cycle set of R9 is simplified to the macro cycle set:

$$\bigcup_{i=1}^{n}[\{A(i).B(i).D(i),C(i).D(i)\}]$$

We believe that the above expression greatly simplifies the task of understanding replicator R9, which may be now read as:

for any i=1,...,n
    do A(i) followed by B(i) followed by D(i),
    or C(i) followed by D(i)

The above macro set is also the macro cycle set of the replicator

R10 #i:1,n,1[(A(i);B(i);D(i)),(C(i);D(i)),@]

The replicator R10 might be slightly easier to read than R9 but R10 involves repeated operation names which make R10 semantically more involved than R9 [LSB79b]. Although we could have defined an "inverse" function of "m-Cycs" to take us from simplified macro cycle objects to macro programs we did not, as this inverse function would in general introduce in macro programs the complexity of repeated operation names. We would not gain anything as the macro objects give us quite a coprehensive reading of the paths in macro programs anyway.

Let us also simplify the macro cycle set of the imbricator R5:

$$\uparrow_{i=1}^{n}[(\{A(i)\} \bullet \downarrow)U\{B(i)\}\bullet\{C(i)\}/(A(n)\})U\{B(n)\}\bullet\{C(n)\}/\{\lambda\}]$$

The first set expression inside "[ ]" may be simplified as follows:

$(\{A(i)\} \circ +) \circ \{C(i)\}) \cup (\{B(i)\} \circ \{C(i)\})$	by rule 1
$(\{A(i)\} \circ +) \circ \{C(i)\}) \cup \{B(i).C(i)\})$	by comp. of "$\circ$"
$(\{A(i)\} \circ +) \circ \{C(i)\}) \cup \{B(i).C(i)\}$	by rule 6
$\{A(i)\} \circ + \circ \{C(i)\}) \cup \{B(i).C(i)\}$	by rule 4

The second expression inside "[ ]" may be simplified as follows:

$\{A(n)\} \cup \{B(n)\} \circ \{C(n)\}$	by rule 6
$\{A(n),B(n)\} \circ \{C(n)\}$	by the composition of "U"
$\{A(n).C(n),B(n).C(n)\}$	by the composition of "$\circ$"

Thus the macro set representing the cycles of R5 may be simplified to

$$\overset{n}{\underset{i=1}{\uparrow}}[(\{A(i)\}\circ+\circ\{C(i)\})\cup\{B(i).C(i)\}/\{A(n).C(n),B(n).C(n)\}/\{\lambda\}]$$

Let finally simplify the macro cycle expression

$$\{SKIP(j)\}\cup(\overset{n}{\underset{i=1}{\cup}}[\{REMOVE(i,j)\}]\circ\{CS\_BEGIN(j)\}\circ\{CS\_END(j)\})$$

which represents the cycles of the last path in the bodyreplicator of MPROG2, namely the path

path SKIP(j),(,[REMOVE(,j)];CSBEGIN(j);CS_END(j)) end

The above set expression may be simplified as follows:

$\{SKIP(j)\}\cup(\overset{n}{\underset{i=1}{\cup}}[\{REMOVE(i,j)\}]\circ\{CS\_BEGIN(j).CS\_END(j)\})$	comp. of "$\circ$"
$\{SKIP(j)\}\cup(\overset{n}{\underset{i=1}{\cup}}[\{REMOVE(i,j)\}\circ\{CS\_BEGIN(j).CS\_END(j)\})]$	rule 5
$\{SKIP(j)\}\cup(\overset{n}{\underset{i=1}{\cup}}[\{REMOVE(i,j)\}.CS\_BEGIN(j).CS\_END(j)\}])$	comp. of "$\circ$"
$\{SKIP(j)\}\cup\overset{n}{\underset{i=1}{\cup}}[\{REMOVE(i,j)\}.CS\_BEGIN(j).CS\_END(j)\}]$	rule 6

To construct the vector firing sequences of a macro program, the ordered cycle sets of the expanded paths will be needed. Let us therefore define the function "cyc-exp" by which macro sets in macro objects are expanded. We will then be in a position to show formally that for a macro program MPROG, generated by the the syntax in appendix D, the relation

Cycles(expand2(MPROG))=cyc-exp(m-Cycs(MPROG))

holds, where the function "Cycles" yields the ordered cycle sets of basic programs and is defined in section 4.1.

The function "cyc-exp" applies to macro cycle objects of macro programs which may be represented by

mcycles BODY-CYCS endmcycles

where BODY-CYCS denotes ordered macro cycle expressions representing the cycle sets of the paths and bodyreplicators in the body of a macro program and may be represented by

BD-CYCS1 &...& BD-CYCSn

where each of BD-CYCSi for i=1,...,n denotes a macro cycle set of a single bodyreplicator or a macro cycle expression representing the cycle set of a path.

A macro cycle set of a bodyreplicator may be represented by

$$\&_{i=1}^{m}[\ \text{BD-CYCS}(i)\ ]$$

where BD-CYCS(i) denotes ordered macro cycle expressions representing the ordered cycle sets of the paths in the regularity of the bodyreplicator. The macro cycle expression representing the cycle set

of a single path may be represented by

pcyc SEQ-CYC endpcyc

where SEQ-CYC denotes the cycle expression representing the cycles of a macro sequence and may be represented by

SP-CYC1 o...o SP-CYCn

where each of SP-CYCi for i=1,...,n denotes the macro cycle expression representing the cycle set of a string produced by "seqpart", which could be a concatenator or distributor generating sequences, a macro orelement or an imbricator generating sequences. The macro cycle set of a concatenator or distributor generating sequences may be represented by

$$\overset{m}{\underset{i=1}{o}}[SEQ\text{-}CYC(i)]$$

where SEQ-CYC(i) denotes the macro cycle expression representing the cycle set of the regularity of concatenators and distributors. The macro cycle expression representing the cycle set of a macro orelement may be represented by

ORP-CYC1 U...U ORP-CYCn

where each of ORP-CYCi for i=1,...,n denotes a macro cycle expression representing the cycle set of a string produced by "orpart", which could be a concatenator or distributor generating orelements, a starelement, or an imbricator generating an orelement. The macro cycle set representing the cycle set of a concatenator or distributor generating orelements may be represented by

$$\overset{m}{\underset{i=1}{U}}[ORP\text{-}CYC(i)]$$

where ORP-CYC(i) denotes the macro cycle expression representing the cycle set of the regularity of the concatenator or distributor. The macro set expression representing the cycle set of a starelement may be represented by

EL-CYC* or EL-CYC

where EL-CYC denotes the macro cycle expression representing the cycle set of an element and may be represented either by

STRING-SET

if the element is an operation, or by

(SEQ-CYC)

if the element is of the form (msequence).

The macro cycle set representing the cycle set of an imbricator may be represented by

$$\uparrow_{i=in}^{n}[SP(i)\downarrow SQ(i)/B/C]$$

where SP(i)↓SQ(i) denotes the macro cycle expression representing the cycle set of the repeatable regularity, B denotes the macro cycle expression representing the cycle set of the innermost regularity and C denotes the macro cycle expression representing the cycle set of the string between the "@"s without its leading and terminating separators. The macro cycle expressions B and C are of form of SEQ-CYC (cf. lemmata 4, 5 in section 3.3). The macro cycle expression SP(i)↓SQ(i) may be represented by

RSEQ-CYC(i)

which may be represented by

RSP-CYC1(i) ●...● RSP-CYCn(i)

where each of RSP-CYCj(i) for j=1,...,n denotes a macro cycle expression of a string produced by "seqpart", except exactly one which either denotes the cycle set of "@t@" represented by "↓", or denotes the macro cycle expression of a string involving "@t@" and it is represented by

ROP-CYC1(i) U...U ROP-CYCn(i)

where each of ROP-CYCj(i) for j=1,...,n denotes a macro cycle expression of a string produced by "seqpart", except exactly one which denotes the macro cycle expression of "@t@" represented by "↓", or the macro cycle expression of a starred element involving "@t@" represented by

REL-CYC(i) or REL-CYC(i)*

where REL-CYC(i) denotes the macro cycle expression of an element involving "@t@" and may be represented by

(RSEQ-CYC(i))

The expansion of macro cycle sets of bodyreplicators, and the macro cycle sets of concatenators and distributors will be defined by iteration, whilst the expansion of macro cycle sets of imbricators will be defined by recursion. For the latter we use the same technique as in "expand2" for defining the expansion of imbricators. Here, we assume the existence of a second stack in which macro cycle sets of imbricators are pushed, while their constituent macro cycle entities in "[ ]" are expanded. We unstack the stacked macro cycle sets when "cyc-exp" is applied to "↓", the expansion of which is considered to be the expansion of the original macro cycle set to which "↓" corresponds, with the lower limit of the index of the macro cycle element increased by one. We associate with this stack two operations, "cyc-push" and "cyc-pop" by which macro cycle sets of imbricators may be pushed in and respectively popped out of the stack.

The function "cyc-exp" is defined by:

cyc-exp(e)=cases e:

1. mcycles BD-CYCS endmcycles → cycles cyc-exp(BD-CYCS)endcycles
2. BD-CYCS1&...&BD-CYCSn → cyc-exp(BD-CYCS1)&...&cyc-exp(BD-CYCSn)
3. $\overset{m}{\underset{i=1}{\&}}$[BD-CYCS(i)] → cyc-exp(BD-CYCS(1))&...&cyc-exp(BD-CYCS(m))
4. pcyc SEQ-CYC endpcyc → cyc-exp(SEQ-CYC)
5. SP-CYC1∘...∘SP-CYCn → cyc-exp(SP-CYC1)∘...∘cyc-exp(SP-CYC)
6. $\overset{m}{\underset{i=1}{\circ}}$[SEQ-CYC(i)] → cyc-exp(SEQ-CYC(1))∘...∘cyc-exp(SEQ-CYC(m))
7. ORP-CYC1 U...U ORP-CYCn → cyc-exp(ORP-CYC1)U...U cyc-exp(ORP-CYCn)
8. $\overset{m}{\underset{i=1}{U}}$[OR-CYC(i)] → cyc-exp(OR-CYC(1))U...U cyc-exp(OR-CYC(m))
9. $\overset{n}{\underset{i=in}{\uparrow}}$[SP(i)↓SQ(i)/B/C] → if n>in then
   cyc-push($\overset{n}{\underset{i=in+1}{\uparrow}}$[SP(i)↓SQ(i)/B/C])
   cyc-exp(SP(in)↓SQ(in))
   if n=in then cyc-exp(B)
   if n>in then cyc-exp(C)
10. RSEQCYC1(k)∘...∘RSEQCYCn(k) → cyc-exp(RSEQCYC1(k))∘...∘cyc-exp(RSEQCYCn(k))
11. RORCYC1(k)U...U RORCYCn(k) → cyc-exp(RORCYC1(k))U...U cyc-exp(RORCYCn(k))
12. RELCYC(k)* → cyc-exp(RELCYC(k))*
13. (RSEQ-CYC(k)) → cyc-exp(RSEQ-CYC(k)
14. ↓ → cyc-exp(cyc-pop)
15. EL-CYC* → cyc-exp(EL-CYC)*
16. (SEQ-CYC) → cyc-exp(SEQ-CYC)
17. STRING-SET → STRING-SET

Let us apply the function "cyc-exp" to expand the macro cycle object of the macro programs MPROG1 and MPROG2. The expansion of the macro cycle object of MPROG1 for n=3 is

```
cycles
  {DEPOSIT(1).REMOVE(1)} &
  {DEPOSIT(2).REMOVE(2)} &
  {DEPOSIT(3).REMOVE(3)} &
  {DEPOSIT(1).DEPOSIT(2).DEPOSIT(3)} &
  {REMOVE(1).REMOVE(3).REMOVE(3)}
endcycles
```

and the expansion of the macro cycle object of MPROG2 for m=3, n=3 is

```
cycles
  {DEPOSIT(1,1).DEPOSIT(2,1).DEPOSIT(3,1)} &
  {REMOVE(1,1).REMOVE(2,1).REMOVE(3,1)}    &
  {SKIP(1),REMOVE(1,1).CS_BEGIN(1).CS_END(1),
   SKIP(1),REMOVE(2,1).CS_BEGIN(1).CS_END(1),
   SKIP(1),REMOVE(3,1).CS_BEGIN(1).CS_END(1) }  &
  {DEPOSIT(1,1).REMOVE(1,1)} &
  {DEPOSIT(2,1).REMOVE(2,1)} &
  {DEPOSIT(3,1).REMOVE(3,1)} &

  {DEPOSIT(1,2).DEPOSIT(2,2).DEPOSIT(3,2)} &
  {REMOVE(1,2).REMOVE(2,2).REMOVE(3,2)}    &
  {SKIP(2),REMOVE(1,2).CS_BEGIN(2).CS_END(2),
   SKIP(2),REMOVE(2,2).CS_BEGIN(2).CS_END(2),
   SKIP(2),REMOVE(3,2).CS_BEGIN(2).CS_END(2) }  &
  {DEPOSIT(1,2).REMOVE(1,2)} &
  {DEPOSIT(2,2).REMOVE(2,2)} &
  {DEPOSIT(3,2).REMOVE(3,2)} &

  {DEPOSIT(1,3).DEPOSIT(2,3).DEPOSIT(3,3)} &
  {REMOVE(1,3).REMOVE(2,3).REMOVE(3,3)}    &
  {SKIP(3),REMOVE(1,3).CS_BEGIN(3).CS_END(3),
   SKIP(3),REMOVE(2,3).CS_BEGIN(3).CS_END(3),
   SKIP(3),REMOVE(3,3).CS_BEGIN(3).CS_END(3) }  &
  {DEPOSIT(1,3).REMOVE(1,3)} &
  {DEPOSIT(2,3).REMOVE(2,3)} &
  {DEPOSIT(3,3).REMOVE(3,3)} &

  {SKIP(1).SKIP(2).SKIP(3),
   SKIP(1).SKIP(2).CS_BEGIN(3).CS_END(3),
   SKIP(1).CS_BEGIN(2).CS_END(2),
   CS_BEGIN(3).CS_END(3)  }
endcycles
```

4.2.3 The Ordered Cycle sets of the Expansion and the Expansion of macro-Cycle Objects of Constrained Macro-Programs

In this section we shall prove that the ordered cycle sets of a basic program obtained by the expansion of a macro program MPROG are the same as the ordered cycle sets obtained by the expansion of the macro cycle objects of MPROG.

THEOREM 4.3:

For any macro program MPROG produced by the constrained syntax rules of appendix D

Cycles(expand2(MPROG))=cyc-exp(m-Cycs(MPROG))

Proof:

We shall prove it by proving the above relation for each case of syntactic entities on which the functions "expand2" and "m-Cycs" apply.

case 1

Applying function "expand2" to a macro program and then "Cycles" we obtain

Cycles(expand2(program MPBODY endprogram))=
Cycles(program expand2(MPBODY)endprogram)=
cycles Cycles(expand2(MPBODY))endcycles

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(program MPBODY endprogram))=
cyc-exp(mcycles m-Cycs(MPBODY)endmcycles)=
mcycles cyc-exp(m-Cycs(MPBODY))endmcycles

The above two expressions are the same provided the relation

Cycles(expand2(MPBODY))=cyc-exp(m-Cycs(MPBODY))

holds, which may be shown by case 2.

case 2

Applying function "expand2" to macro program body and then "Cycles" we obtain

Cycles(expand2(CPBR1...CPBRn))=
Cycles(expand2(CPBR1)...expand2(CPBRn))=
Cycles(expand2(CPBR1))&...&Cycles(expand2(CPBRn))

The last step is justified as

expand2(CPBRi) for any i=1,...,n

yields a collection of basic paths and

Cycles(CP1) & Cycles(CP2)=Cycles(CP1 CP2)

where CP1 and CP2 are collections of basic paths. To show the above relation let us define CP1 and CP2 as follows:

CP1=$P^1 1...P^1 m$
CP2=$P^2 1...P^2 k$

where $P^1 i$ and $P^2 j$ for i=1,...,m and j=1,...,k are basic paths. Then the relation

Cycles($P^1 1$ $P^1 2...P^1 m$) & Cycles($P^2 1$ $P^2 2...P^2 k$)=
Cyc($P^1 1$)&Cyc($P^1 2$)&...&Cyc($P^1 m$)&Cyc($P^2 1$)&Cyc($P^2 2$)&...&Cyc($P^2 k$) =
Cycles($P^1 1$ $P^1 2...P^1 m$ $P^2 1$ $P^2 2...P^2 k$)

holds. Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(CPBR1...CPBRn))=
cyc-exp(m-Cycs(CPBR1)&...&m-Cycs(CPBRn))=
cyc-exp(m-Cycs(CPBR1))&...&cyc-exp(m-Cycs(CPBRn))

The above expressions are the same provided the relation

Cycles(expand2(CPBRi))=cyc-exp(m-Cycs(CPBRi))

holds for any i=1,...,n, which may be shown by case 3.

case 3

Applying function "expand2" to a single path or bodyreplicator possibly headed by collectivisors and then "Cycles" we obtain

Cycles(expand2(COLs PBR))=
Cycles(expand2(PBR))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(COLs PBR))=
cyc-exp(m-Cycs(PBR))

The above expressions are the same as may be shown by cases 4 and 6, depending on whether PBR denotes a bodyreplicator or a macro path.

case 4

Applying function "expand2" to a bodyreplicator and then "Cycles" we obtain

Cycles(expand2(#i:1,n,1[PBRs(i)]))=
Cycles(expand2(PBRs(1))...expand2(PBRs(n)))=
Cycles(expand2(PBRs(1)))&...&Cycles(expand2(PBRs(n)))

Since the expansion of PBRs(i)

expand2(PBRs(i)) for any i=1,...,n

yields a collection of basic paths, the last step is justified as shown in case 2.

Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(#i:1,n,1[PBRs(i)]))=

$$\text{cyc-exp}\Big(\&_{i=1}^{n}[\text{m-Cycs}(\text{PBRs}(i))]\Big)=$$

cyc-exp(m-Cycs(PBRs(1)))&...&cyc-exp(m-Cycs(PBRs(n)))

The above expressions are the same provided the relation

Cycles(expand2(PBRs(i)))=cyc-exp(m-cycs(PBRs(i)))

holds for any i=1,...,n, which may be shown by case 5.

case 5

Applying function "expand2" to a collection of paths and bodyreplicators and then "Cycles" we obtain

Cycles(expand2(PBR1...PBRn))=
Cycles(expand2(PBR1)...expand2(PBRn))=
Cycles(expand2(PBR1))&...&Cycles(expand2(PBRn))

Since the expansion of PBRi

expand2(PBRi) for any i=1,...,n

yields a collection of basic paths, the last step is justified as was shown in case 2.

Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(PBR1...PBRn))=
cyc-exp(m-Cycs(PBR1)&...&m-Cycs(PBRn))=
cyc-exp(m-Cycs(PBR1))&...&cyc-exp(m-Cycs(PBRn))

The above expressions are the same provided the relation

Cycles(expand2(PBRi))=cyc-exp(m-Cycs(PBRi))

holds for any i=1,...,n, which may be shown by cases 4 and 6, depending on whether a PBRi for any i=1,...,n is a bodyreplicator or a macro path.

case 6

Applying function "expand2" to a macro path and then "Cycles" we obtain

Cycles(expand2(path MSEQ end))=
Cyc(expand2(path MSEQ end))=
Cyc(path expand2(MSEQ) end)=
Cyc(expand2(MSEQ))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(path MSEQ end))=
cyc-exp(pcyc m-Cycs(MSEQ) endpcyc)=
cyc-exp(m-Cycs(MSEQ))

The above expressions are the same as may be shown by case 7.

case 7

Applying function "expand2" to a macro sequence and then "Cyc" we obtain

Cyc(expand2(SEQPRT1;...;SEQPRTn))=
Cyc(expand2(SEQPRT1);...;expand2(SEQPRTn))=
Cyc(expand2(SEQPRT1))∘...∘Cyc(expand2(SEQPRTn))

The last step is justified as

expand2(SEQPRTi)) for any i=1,...,n

is a basic sequence (cf. case 2 of theorem 4.1). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(SEQPRT1;...;SEQPRTn))=
cyc-exp(m-Cycs(SEQPRT1)∘...∘m-Cycs(SEQPRTn))=
cyc-exp(m-Cycs(SEQPRT1))∘...∘cyc-exp(m-Cycs(SEQPRTn))

The above expressions are the same provided the relation

Cyc(expand2(SEQPRTi))=cyc-exp(m-Cycs(SEQPRTi))

holds for any i=1,...,n, which may be shown by cases 8, 9, 10 and 13, depending on whether a SEQPRTi for $1 \leq i \leq n$, is a concatenator, a distributor generating sequences, a macro orelement, or an imbricator generating sequences respectively.

case 8

Applying function "expand2" to a concatenator generating sequences and then "Cyc" we obtain

Cyc(expand2(#i:1,n,1[MSEQ(i);@]))=
Cyc(expand2(MSEQ(1));...;expand2(MSEQ(n)))=
Cyc(expand2(MSEQ(1)))∘...∘Cyc(expand2(MSEQ(n)))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(#i:1,n,1[MSEQ(i);@]))=

$$\text{cyc-exp}\left(\overset{n}{\underset{i=1}{\circ}}[\text{m-Cycs}(\text{MSEQ}(i))]\right)=$$

cyc-exp(m-Cycs(MSEQ(1)))∘...∘cyc-exp(m-Cycs(MSEQ(n)))

The above expressions are the same provided the relation

Cyc(expand2(MSEQ(i)))=cyc-exp(m-Cycs(MSEQ(i)))

holds for any i=1,...,n, which may be shown by case 7 as MSEQ(i) for i=1,...,n is a macro sequence.

case 9

Applying function "expand2" to a distributor generating sequences and then "Cyc" we obtain

Cyc(expand2(;[MSEQ]))=

Cyc(expand2(MSEQ(1));...;expand2(MSEQ(n)))

Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(;[MSEQ]))=

$$\text{cyc-exp}\left(\bullet_{i=1}^{n}[\text{m-Cyc}(\text{MSEQ}(i))]\right)$$

From this point the proof follows as for case 8.

case 10

Applying function "expand2" to a macro orelement and then "Cyc" we obtain

Cyc(expand2(ORPRT1,...,ORPRTn))=

Cyc(expand2(ORPRT1),...,expand2(ORPRTn))=

Cyc(expand2(ORPRT1))U...U Cyc(expand2(ORPRTn))

The last step is justified as

expand2(ORPRTi) for any i=1,...,n

is a basic orelement (cf. case 5 of theorem 4.2). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(ORPRT1,...,ORPRTn))=

cyc-exp(m-Cycs(ORPRT1)U...U m-Cycs(ORPRTn))=

cyc-exp(m-Cycs(ORPRT1))U...U cyc-exp(m-Cycs(ORPRTn))

The above expressions are the same provided the relation

Cyc(expand2(ORPRTi))=cyc-exp(m-Cycs(ORPRTi))

holds for any i=1,...,n which may be shown by cases 11, 12, 13, 21, 22, 23 depending on whether ORPTRi for $1 \leq i \leq n$ is a concatenator generating orelements, a distributor generating orelements, an imbricator generating orelements, a macro starelement, an operation or an element of the form (msequence), respectively.

case 11

Applying function "expand2" to concatenators generating orelements and then "Cyc" we obtain

Cyc(expand2(#i:1,n,1[MOR(i),@]))=
Cyc(expand2(MOR(1)),...,expand2(MOR(n)))=
Cyc(expand2(MOR(1)))U...U Cyc(expand2(MOR(n)))

The last step is justified as

expand2(MOR(i)) for any i=1,...,n

is a basic orelement (cf. case 5 of theorem 4.2). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(#i:1,n,1[MOR(i),@]))=

$$\text{cyc-exp}\left(\bigcup_{i=1}^{n}[\text{m-Cycs}(\text{MOR}(i))]\right)=$$

cyc-exp(m-Cycs(MOR(1)))U...U cyc-exp(m-Cycs(MOR(n)))

The above expressions are the same provided the relation

Cyc(expand2(MOR(i)))=cyc-exp(m-Cycs(MOR(i)))

holds for any i=1,...,n, which may be shown by case 10 as MOR(i) for i=1,...,n is a macro orelement.

case 12

Applying function "expand2" to distributors generating orelements and then "Cyc" we obtain

Cyc(expand2(,[MOR]))=

Cyc(expand2(MOR(1)),...,expand2(MOR(n)))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(,[MOR]))=

$$\text{cyc-exp}(\overset{n}{\underset{i=1}{U}}[\text{m-Cyc}(\text{MOR}(i))])=$$

From this point the proof follows as for case 11.

case 13

Applying function "expand2" to an imbricator and then "Cyc" we obtain

Cyc(expand2(#i:in,n,1[p(i) @t@ q(i)]))=

Cyc(if in>n then expand2(t')
    if in=n then expand2(p'(n) t q'(n))
    if in<n then expand2(p(in) @t@ q(in)))=

| if in>n then Cyc(expand2(t'))
| if in=n then Cyc(expand2(p'(n) t q'(n)))
| if in<n then Cyc(expand2(p(in) @t@ q(in)))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(#i:in,n,1[p(i) @t@ q(i)]))=

$$\text{cyc-exp}(\overset{n}{\underset{i=in}{\uparrow}}[\text{m-Cycs}(p(i)\ @t@\ q(i))/\text{m-Cycs}(p'(n)\ t\ q'(n))/\text{m-Cycs}(t')])=$$

| if in>n then cyc-exp(m-Cycs(t'))
| if in=n then cyc-exp(m-Cycs(p'(n) t q'(n)))
| if in<n then cyc-exp(m-Cycs(p(in) @t@ q(in)))

We now have to show that the three relations hold:

if in>n then
Cyc(expand2(t'))=exp-cyc(m-Cycs(t'))

if in=n then
Cyc(expand2(p'(n) t q'(n)))=exp-cyc(m-Cycs(p'(n) t q'(n)))

if in<n then
Cyc(expand2(p(in) @t@ q(in)))=exp-cyc(m-Cycs(p(in) @t@ q(in)))

Depending on whether the imbricator generates a sequence, an orelement, a starelement or an element, the strings "t'" and "p'(n) t q'(n)" will be sequences, orelements, starelements or elements, respectively. Therefore, for in>n and in=n the first two relations may be shown to hold by cases 7, 10, 21, 23 respectively.

We have still to prove the third relation

Cyc(expand2(p(in) @t@ q(in)))=cyc-exp(m-Cycs(p(in) @t@ q(in)))

when in<n. In this case before "expand2" and "cyc-exp" are applied the imbricator

#i:in+1,n,1[p(i) @t@ q(i)]

is stacked into the imbricator stack and the macro cycle set

$$\uparrow_{i=in+1}^{n}[\text{m-Cycs}(p(i)\ @t@\ q(i))/\text{m-Cycs}(p'(n)\ t\ q'(n))/\text{m-Cycs}(t')]=$$

into the macro cycle set stack. Observe that the above macro cycle set is the macro cycle set of the imbricator stacked into the imbricator stack. We shall use this fact in case 20.

Depending on whether the imbricator is genuine or not and whether it generates a sequence, an orelement, a starelement or an element, the string "p(in) @t@ q(in)" may be of six forms F1 to F6. For each form Fi $1 \leq i \leq 6$ the relation

Cyc(expand2(Fi))=cyc-exp(m-Cycs(Fi))

must hold. The six forms and the corresponding cases by which the above relation may be shown to hold are as follows:

F1.	IM_SP1(in);...;IM_SPn(in)	case 14
F2.	AT_SP1(in);...;AT_SPn(in)	case 18
F3.	IM_OP1(in);...;IM_OPn(in)	case 15
F4.	AT_OP1(in),...,AT_OPn(in)	case 19
F5.	IM_EL(in)*	case 16
F6.	(IMBR_SEQ(in))	case 17

case 14

Applying function "expand2" to the string inside "[ ]" of a genuine sequence imbricator or to the string produced by the non-terminals "orimbrseq", "starimbrseq", "elimbrseq" and then "Cyc" we obtain

Cyc(expand2(IM_SP1(k);...;IM_SPn(k)))=
Cyc(expand2(IM_SP1(k));...;expand2(IM_SPn(k)))=
Cyc(expand2(IM_SP1(k)))∘...∘Cyc(expand2(IM_SPn(k)))

The last step is justified as

expand2(IM_SPi)) for any i=1,...,n

is a basic sequence (cf. case 2 of theorem 4.1). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(IM_SP1(k);...;IM_SPn(k)))=
cyc-exp(m-Cycs(IM_SP1(k))∘...∘m-Cycs(IM_SPn(k)))=
cyc-exp(m-Cycs(IM_SP1(k)))∘...∘cyc-exp(m-Cycs(IM_SPn(k)))

The above expressions are the same provided the relation

Cyc(expand2(IM_SPi(k)))=cyc-exp(m-Cycs(IM_SPi(k)))

holds for all i=1,...,n. The above relation may be shown to hold by cases 8, 9, 13 when IM_SPi(k) for $1 \leq i \leq n$ is a concatenator, distributor and imbricator respectively generating sequences, by case 10 when IM_SPi(k) is an orelement, or by case 20 when IM_SPi(k) is the string "@t@".

case 15

Applying function "expand2" to the string inside a genuine imbricator generating orelements or to the string produced by one of the non-terminals "seqimbror", "orimbr_in_or", "starimbror", "elimbror" and then "Cyc" we obtain

Cyc(expand2(IM_OP1(k),...,IM_OPn(k)))=
Cyc(expand2(IM_OP1(k)),...,expand2(IM_OPn(k)))=
Cyc(expand2(IM_OP1(k)))U...U Cyc(expand2(IM_OPn(k)))

The last step is justified as

expand2(IM_OPi) for any i=1,...,n

is a basic orelement (cf. case 5 of theorem 4.2). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(IM_OP1(k),...,IM_OPn(k)))=
cyc-exp(m-Cycs(IM_OP1(k))U...U m-Cycs(IM_OPn(k)))=
cyc-exp(m-Cycs(IM_OP1(k)))U...U cyc-exp(m-Cycs(IM_OPn(k)))

The above expressions are the same provided the relation

Cyc(expand2(IM_OPi(k)))=cyc-exp(m-Cycs(IM_OPi(k)))

for i=1,...,n holds. The above relation may be shown to hold by cases 11, 12, 13 when IM_OPi(k) for $1 \leq i \leq k$ is a concatenator or distributor or imbricator generating orelements, by case 21 if it is a starred element, by cases 22 or 23 if it is an operation or an element of the form "(msequence)" respectively, by case 16 if it is a starred element involving "@t@", and finally by case 17 if it an element involving "@t@".

case 16

Applying function "expand2" to strings produced by the non-terminals "seqimbrstarel", "orimbrstarel", "starimbrstarel" and "elimbrstarel", and then "Cyc" we obtain

Cyc(expand2(IM_EL(k)*))=
Cyc(expand2(IM_EL(k))*)=
Cyc(expand2(IM_EL(k)))*

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(IM_EL(k)*))=
cyc-exp(m-Cycs(IM_EL(k))*)=
cyc-exp(m-Cycs(IM_EL(k)))*

The above expressions are the same provided the relation

Cyc(expand2(IM_OPi(k)))=exp-cyc(m-Cycs(IM_OPi(k)))

holds, which may be shown by case 17.

case 17

Applying function "expand2" to the strings represented by the syntactic entity (IMBR_SEQ(k)), in which the syntactic variable represents strings produced by the non-terminals "seqimbrseq", "orimbrseq", "starimbrseq", "elimbrseq", and then "Cyc" we obtain

Cyc(expand2((IMBR_SEQ(k)))=
Cyc((expand2(IMBR_SEQ(k)))=
Cyc(expand2(IMBR_SEQ(k))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs((IMBR_SEQ(k)))=
cyc-exp((m-Cycs(IMBR_SEQ(k)))=
cyc-exp(m-Cycs(IMBR_SEQ(k))

The above expressions are the same as may be shown by case 14 if the "@t@" is further nested inside "( )" and by case 18 if the "@t@" is not further nested inside "( )". In the special case of an imbricator generating sequences the string "@t@" may only appear as an element between two semicolons and the equality of the above expressions may be shown by case 20.

case 18

Applying function "expand2" to the sequence involving "@t@" in one of its orelements, and then "Cyc" we obtain

Cyc(expand2(AT_SP1(k);...;AT_SPn(k)))=
Cyc(expand2(AT_SP1(k));...;expand2(AT_SPn(k)))=
Cyc(expand2(AT_SP1(k)))o...oCyc(expand2(AT_SPn(k)))

The last step is justified as

expand2(AT_SPi)) for any i=1,...,n

is a basic sequence (cf. case 2 of theorem 4.1). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(AT_SP1(k);...;AT_SPn(k)))=
cyc-exp(m-Cycs(AT_SP1(k))o...om-Cycs(AT_SPn(k)))=
cyc-exp(m-Cycs(AT_SP1(k)))o...ocyc-exp(m-Cycs(AT_SPn(k)))

The above expression are the same provided the relation

Cyc(expand2(AT_SPi(k)))=cyc-exp(m-Cycs(AT_SPi(k)))

holds for all i=1,...,n. The above relation may be shown to hold by cases 8, 9, 13 if AT_SPi(k) for $1 \leq i \leq n$ is a concatenator, a distributor or an imbricator respectively, generating sequences, by case 10 if it is

an orelement or by case 19 if it invoves "@t@" as one of its elements.

case 19

Applying function "expand2" to an orelement involving "@t@" as one of its elements and then "Cyc" we obtain

Cyc(expand2(AT_OP1(k),...,AT_OPn(k)))=
Cyc(expand2(AT_OP1(k)),...,expand2(AT_OPn(k)))=
Cyc(expand2(AT_OP1(k)))U...U Cyc(expand2(AT_OPn(k)))

The last step is justified as

expand2(AT_OPi) for any i=1,...,n

is a basic orelement (cf. case 5 of theorem 4.2). Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(AT_OP1(k),...,AT_OPn(k)))=
cyc-exp(m-Cycs(AT_OP1(k))U...U m-Cycs(AT_OPn(k)))=
cyc-exp(m-Cycs(AT_OP1(k)))U...U cyc-exp(m-Cycs(AT_OPn(k)))

The above expressions are the same provided the relation

Cyc(expand2(AT_OPi(k)))=cyc-exp(m-Cycs(AT_OPi(k)))

holds for all i=1,...,n. The above relation may be shown to hold by cases 11, 12, 13 when AT_OPi(k) for $1 \leq i \leq n$ is a concatenator or a distributor or an imbricator generating orelements respectively, by cases 16 and 17 if the "@t@" is further nested inside "( )", or by case 20 if AT_OPi(k) for $1 \leq i \leq n$ is "@t@".

case 20

Applying function "expand2" to the special element "@t@" and then "Cyc" we obtain

Cyc(expand2(AT_EL))=
Cyc(expand2(imbr-pop))

where imbr-pop returns the imbricator at the top of the stack which will be denoted by Imbr. Applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(AT_EL))=
cyc-exp(⍬)=
cyc-exp(cyc-pop)

where cyc-pop returns the macro cycle set at the top of the stack which will be denoted by MCset. Since MCset is the same as m-Cycs(Imbr) (cf. case 13) for the above expressions to be the same the relation

Cyc(expand2(Imbr))=cyc-exp(m-Cycs(Imbr))

must hold, which may be shown by case 13.

case 21

Applying function "expand2" to a starelement and then "Cyc" we obtain

Cyc(expand2(EL*))=
Cyc(expand2(EL)*)=
Cyc(expand2(EL))*

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(EL*))=
cyc-exp(m-Cycs(EL)*)=
cyc-exp(m-Cycs(EL))*

The above two expressions are the same provided the relation

Cyc(expand2(EL))=cyc-exp(m-Cycs(EL))

holds, which may be shown by cases 22 or 23 when EL is an operation or

an element of the form (MSEQ) respectively.

case 22

Applying function "expand2" to an operation and then "Cyc" we obtain

Cyc(expand2(OP))=
Cyc(OP)=
{OP}

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs(OP))=
cyc-exp({OP})=
{OP}

which are the same.

case 23

Applying function "expand2" to an element of the form (MSEQ) and then "Cyc" we obtain

Cyc(expand2((MSEQ)))=
Cyc((expand2(MSEQ)))=
Cyc(expand(MSEQ))

and applying "m-Cycs" first and then "cyc-exp" we obtain

cyc-exp(m-Cycs((MSEQ)))=
cyc-exp((m-Cycs(MSEQ)))=
cyc-exp(m-Cycs(MSEQ))

The above expressions may be shown to be the same by case 7.

As we have proven the theorem for all possible cases of syntactic entities of macro programs on which the functions "expand2" and "m-Cycs" apply, we may conclude that the theorem holds for all macro programs

produced by the syntax of appendix D.///

In this chapter we examined two methods by which the vector firing sequences of basic programs generated from macro programs may be derived from the macro programs themselves. We first reduced the problem of finding the vector firing sequences to the problem of finding the ordered cycle sets of the basic programs directly from the macro programs generating them.

The first method for finding the ordered cycle sets involves two steps. In the first step all bodyreplicators are expanded and ordered expressions are derived which yield the cycle sets of pure macro paths. In the second step the cycle sets of basic paths are derived directly from the pure macro paths which generate them, by expanding parts of macro sequences and constructing their cycle sets of the strings they generate, which are then composed together by union and concatenation operations. What the meaningful smallest parts of macro sequences makes sense to expand and to find their cycle sets depends on the kind of strings macro elements generate. If the macro elements generate syntactically strong strings, then the smallest such parts are the elements and the macro elements of the sequences. as we have demonstrated in section 4.1.2. If the macro elements do not generate syntactically strong strings but matching pairs of parentheses, as macro elements in the notation of section 3.2, then the smallest such parts are the orelements involving starelements.

The second method for finding the ordered cycle sets may be applied to programs the macro elements of which generate syntactically strong regularities. According to this method macro cycle objects are constructed from macro programs which concisely represent and precisely generate upon expansion the ordered cycle sets of the basic program generated by the expansion of the macro programs.

The second method has an advantage over the first. The ordered cycle sets obtained by the first method are those of a basic program generated by a macro program with all its integer constant parameters given specific values. But according to the second method, macro cycle objects may also involve parameters. Of course upon the expansion of

macro cycle objects all integer constants should be given specific values. The parametarized representation of ordered cycle sets by macro cycle objects is very important in the verification of parametarized macro programs, where we frequently need to argue in terms of the cycle sets of basic paths generated by macro paths. Macro cycle objects give the formal basis for lucid and precise arguments on the cycles of paths.

The second method has the disadvantage that it may only be applied to constrained macro programs, which in general are not as concise as macro programs in the notation of 3.2. In other words a macro program in the notation of 3.2 generating the same basic program as a macro program in the notation of 4.2.1 is in general more concise than the latter.

5 CONCLUSIONS

In this thesis we were mainly concerned with the macro COSY notation. We re-examined and revised all aspects of the macro COSY notation, its design as a specification language for asynchronous systems, the formal syntax for macro programs, the expansion rules for macro elements and for complete macro programs. Various previously developed notations and subnotations and their formal syntax were carefully examined and their advantages and disadvantages were pointed out.

In the process of programming with these notations we came to formulate better the properties a "good" macro notation should possess:

1. The syntactic well-formedness of a macro program should imply that its expansion yields a syntactically well-formed basic program.

2. The notation should allow the generation of a large class of basic programs, and their concise representation.

3. The syntax for macro COSY programs should be uniform with the syntax for basic COSY programs.

4. The reading of macro programs should be possible without formal expansion.

Previously developed macro notations do not in general possess all four of these properties. The syntax rules for most of them permit macro programs which do not expand to basic programs and meta-restrictions are introduced to eliminate these "wide" programs. The syntax rules are not uniform with syntax rules for basic programs and are not complete as replicators in collectivisors are not given formal syntax rules.

In designing the new macro notation we adopted the same types of constructs, that is collectivisors, replicators and distributors, for representing and generating basic programs, as in previously developed notations. In the new notation, we even incorporated and combined aspects from various notations, contributing to the generality and readability of macro programs. These aspects include the

bodyreplicators generating paths and processes and permitting nesting of other bodyreplicators, the implicit and explicit lowerbound of dimensions of rectangular arrays, etc. We have made a number of modifications, improving the readability of macro programs, such as the addition of the terminal symbol "endarray", the change of the form and the position of the index specification part of replicators and the change of the round parentheses in distributors to square ones.

Apart from the above modifications others more fundamental to the design of the notation were made. Replicators in collectivisors were carefully designed in relation to distributors. Replicators in collectivisors permit subscripted operations which correspond to arrays, not necessarily rectangular. More general shapes of arrays could be permitted by replicators but we have restricted them in order to keep the expansion of distributors relatively simple. Replicators in sequences were designed to generate sequences. We have excluded all other replicators such as the range, context and neighbourhood dependent replicators, which are permitted in some other notations.

Finally, a number of extensions improving the generality of macro programs were developed. These extensions include the part of the imbricators between the two "@"s, appearing only once in the strings obtained by their expansion, left and right replicators which are permitted to expand to empty strings, and a number of extensions of distributors: the relaxation of the compatibility of distributable dimensions, the generalization of the strings they may generate and in particular the symmetric nesting of replicators and distributors, the subrange and the selection of distributable dimensions features.

The formal syntax of programs in this notation is according to our requirements close-fitting, since we have avoided the use of meta-restriction rules constraining the regularities replicators generate as in previous notations. The restrictions we have imposed on macro programs are of a context sensitive nature. The expansion of macro programs was formally defined by the function "expand" in section 3.3.3. In this function we used three auxilliary functions, "replexp^0", "distrexp^0" and "gelexp^0" defining the first order expansion of replicators, distributors and of the left and right replicators in generalized

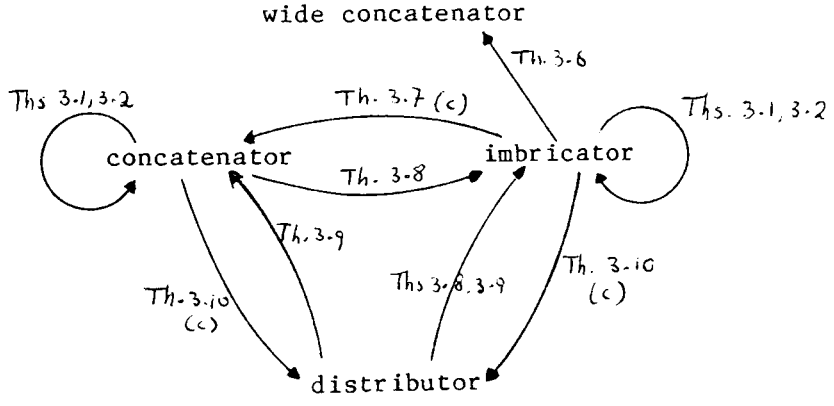
elements respectively. The expansions of all replicators and distributors were formally defined in terms of the premitive recursive operator "COPY". The expansion of distributors was directly defined. We proved by theorems 3.3 and 3.4 that the expansion of concatenators and respectively imbricators yields macro sequences. Theorem 3.5 shows that if all left and right replicators of a generalized element are expanded the resulting string is a macro sequence, and the corollary of theorem 3.9 that the expansion of distributors are also macro sequences. We have used the above theorems 3.3, 3.4, 3.5 and the corollary of 3.9 in theorem 3.11 proving that the expansion of complete macro programs produced by the syntax rules in section 3.2 yields well-formed basic programs. These theorems show the close fittness of the syntax rules.

The syntax rules are also uniform with the rules for basic COSY, since analogous constructs in both notations are expressed by similar rules and various constructs of macro COSY were expressed in a style similar to basic COSY.

The development of syntax rules for macro programs in the new notation did not possess any difficulties, apart from the syntax rules for imbricators. The syntax of imbricators and in particular genuine imbricators was the reason meta-restriction rules had to be used. The problems were twofold: to express that any number of opening parentheses on the left of "@t@" should match with closing parentheses on the right of "@t@" and to express it in a manner similar to basic COSY. Four groups of syntax rules were developed. The first group (CFm) gave context-free rules but specified a mixed precedence of ";" and ",". The second group (CS) gave context-sensitive rules. The third (CFr) gave context-free rules specifying the precedence of "," over ";" but required meta-restriction rules to exclude strings involving more than two "@"s. Only the fourth group (CF) involving close fitting context-free rules specifying the precedence of "," over ";" satisfied our requirements.

We proved a number of theorems which give the relation between concatenators, imbricators and distributors. We showed by theorems 3.1 and 3.2 that for any replicator there exist a whole family of replicators expanding to the same string as the former. We showed how a

replicator may be transformed into its normal form and how from a replicator in normal form all other replicators in the same family may be obtained. We showed by theorem 3.7 that certain imbricators, the non-genuine imbricators, could be replaced by concatenators. By the corollary of theorem 3.7 the syntax rules for imbricators could be restricted to permit only genuine imbricators without restricting the generality of the notation. As it was demonstrated, this choice would jeopardize the conciseness and readability of macro programs. We showed by theorem 3.8 that all concatenators could be replaced by imbricators. A corollary of this theorem is that imbricators are sufficient and concatenators could be eliminated altogether. But as programs would not be as concise and readable this option was rejected as well. Theorem 3.7 showed that all distributors could be replaced by concatenators and following theorem 3.8 by imbricators also. Theorem 3.8 gave the conditions under which concatenators and imbricators may be replaced by distributors. We also proved theorem 3.6 which showed that wide concatenators are sufficient to generate any strings our imbricators generate. We only indicated in section 3.4 how wide concatenators could be modified to form macro elements permitted by our notation, but no formal result or method was obtained regarding this direction. The following figure shows possible replacements of sequence replicators, distributors and wide concatenators:



In the above figure arcs from A to B indicate possible replacement of macro elements of type A by type B. Arcs are labelled by the relevant theorem. A "(c)" labelling an arc indicates that the replacement is only possible under certain conditions.

In chapter 4 we explored ways by which the semantics of basic programs generated from macro programs may be directly obtained from the macro programs themselves. The basis for the construction of vector firing sequences directly from macro programs is that the set of vector operations and the set of firing sequences of basic paths may be obtained from the ordered cycle sets of paths of a basic program, and that these ordered cycle sets may be constructed directly from the macro programs. Two methods were developed for constructing the ordered cycle sets of basic programs directly from the macro programs which generate the basic programs.

The first method may be applied to macro programs in any macro notation, as long as the cycles of syntactically strong strings are constructed. We demonstrated the method for two macro notations: The notation in section 3.2 in which the syntactically strong strings are its orelements consisting of starelements, and the restrictive notation of section 4.1.2 in which the smaller syntactically strong strings are its elements or macro elements. Although the latter notation is less general than that of section 3.2 it is much more readable.

The second method may only be applied to macro programs the macro elements of which generate syntactically strong regularities. The syntax for such constrained programs was developed in section 4.2.1. Programs in this notation have the disadvantage that they are less general and less concise than programs in the notations of sections 3.2 and 4.1.2 but they have the advantage that they are much more easily readable. They also possess the advantage that they allow the parametarized representation of ordered cycle sets by the macro cycle objects which is of a primary importance in the verification of parametarized macro programs.

6 REFERENCES

[A77] AGERWALA T. : Some Extended Semaphore Primitives. Acta Informatica 8, pp. 201-220, 1977.

[A69] ARBIB M.A. : Theories of Abstract Automata. Prentice Hall, Inc., 1969.

[B79] BACKHOUSE R.C. : Syntax of Programming Languages: Theory and Practice. ed. C.A.R. Hoare, Prentice Hall International, 1979.

[B79] BEST E. : Adequacy of Path Programs. In: Net Theory and Applications: Proc. of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979 (Ed. W Brauer). Lecture Notes in Computer Science 84, Springer Verlag, 1980.

[B82] BEST E. : Adequacy Properties of Path Programs, Theoretical Computer Science 18 (1982), pp 149-171, North-Holland Publishing Co.

[BM41] BIRKHOFF G., MacLANE S. : A Survey of Modern Algebra. MacMillan Co., New York, 1941.

[BH73] BRINCH HANSEN P. : Operating System Principles. Prentice Hall, New Jersey, 1973.

[BH74] BRINCH HANSEN P. : Concurrent Pascal: A Programming Language for Operating System Design. TR No. 10, California Institute of Technology, April 1974.

[BH75] BRINCH HANSEN P. : The SOLO Operating System. California Institute of Technology, 1975.

[B67] BURSTALL R.M. : Proving Properties of Programs by Structural Induction. Computer Journal, 12, Number 1, pp. 41-47.

[C76] CAMPBELL R.H. : Path Expressions: A technique for specifying process synchronization. Ph.D. Thesis, Computing Laboratory, University of Newcastle-upon-Tyne, 1976.

[CH74] CAMPBELL R.H., HABERMANN A.N. : The Specification of process synchronization by path expressions. Lecture Notes in Computer Science 16, Springer Verlag, 1974.

[CL75] CAMPBELL R.H., LAUER P.E. : A Hierarchy of Buffer Systems in a parallel processing environment. MRM/88, Computing Laboratory, University of Newcastle-upon-Tyne, May 1975.

[C80] COTRONIS J.Y. : Defining Priorities of Activations of Operations in Conflict in COSY. ASM/73, 1980, Computing Laboratory, University of Newcastle-upon-Tyne.

[CL79] COTRONIS J.Y., LAUER P.E. : On Definitions of various Notions of Monitors in the COSY Notation. ASM/58 Computing Laboratory, University of Newcastle-upon-Tyne, March 1979.

[CL81] COTRONIS J.Y., LAUER P.E. : A New Macro COSY Notation and Grammar. ASM/81, Computing Laboratory, University of Newcastle-upon-Tyne, November 81.

[CG77] COURTOIS P.J., GEORGES J, : On Starvation Prevention. R.A.I.R.O. Informatique/Computer Science 11, pp. 127-141, 1977.

[D77] DEVILLERS R. : Non starving solutions for the dining philosopher's problem. ASM/30, Computing Laboratory, University of Newcastle-upon-Tyne, June 1977.

[D79a] DEVILLERS R. : A simplified Hyperfast Banker. ASM/56, Computing Laboratory, University of Newcastle-upon-Tyne, January 1979.

[D79b] DEVILLERS R. : On the Banker with Several Currencies. ASM/57, Computing Laboratory, University of Newcastle-upon-Tyne, February 1979.

[DL77] DEVILLERS R., LAUER P.E. : Some Solutions for the Reader/Writer Problem. ASM/31, Computing Laboratory, University of Newcastle-upon-Tyne, June 1977.

[D68] DIJKSTRA E. W. : Cooperating Sequential Processes. In: Programming Languages, ed. F. Genuys, Academic Press, New York, 1968.

[D76] DIJKSTRA E. W. : A Discipline of Programming. Prentice Hall, 1976.

[G71] GRIES D. : Compiler Construction fot Digital Computers. John Wiley and Sons, Inc., 1971.

[H75] HABERMANN A.N. : Path Expressions. Carnegie-Mellon University, Pittsburgh, June 1975.

[HW60] HARDY G.H., WRIGHT E.M : An Introduction to the Theory of Numbers. Oxford: Clarendon Press, 1960.

[H65] HARRISON M.A. : Introduction to Switching and Automata Theory. McGraw Hill Company, 1960.

[H72] HOARE C.A.R. : Towards a Theory of Parallel Processing. In: Operating Systems Techniques, pp. 61-72, Academic Press, New York, 1972.

[H73] HOARE C.A.R. : A structured paging system. CACM Vol. 16, 1973, Number 3, pp. 209-215.

[H74] HOARE C.A.R. : Monitors: An Operating System Structuring Concept. CACM Vol. 17, 1974, Number 10, pp. 549-557.

[H80] HOARE C.A.R. : Communicating Sequential Processes. In: On the construction of programs. (Eds. McKeag and MacNaghten) Cambridge University Press 1980.

[HC70] HOLT A.W., COMMONER P. : Events and Conditions. Applied Data Research, New York, 1970.

[HU79] HOPCROFT J.E., ULLMAN J.D. : Introduction to Automata Theory, Languages and Computation. Addis Wesley, 1979.

[J81] JANICKI R. : On the Design of Concurrent Schemes. Procedings of the 2nd Conference on Distributed Computing Systems, Paris, 1981. IEEE Press.

[KM69] KARP R. M., MILLER R. E. : Parallel Program Schemata. JCSS Vol 3, pp. 147-195, 1969.

[L76] LAUER P.E. : Toward a system specification language based on paths and processes PART I: The Notation. ASM/19, Computing Laboratory, University of Newcastle-upon-Tyne, 1979.

[L79] LAUER P.E. : COSY Subnotations: Replicators and Basic Notation, Part 4. ASM/62, Computing Laboratory, University of Newcastle-upon-Tyne, June 1979.

[L81] LAUER P.E. : Synchronization of Concurrent Processes without Globality Assumptions, SIGPLAN Notices, Vol 16, No 9, Sept 1981. An expanded version is included in the book: 1 New Advances in Distributed Computer Systems, pages 341-365, Reidel Pub. Co., NATO Advanced Study Institute Series C80.

[LBS77] LAUER P.E., BEST E., SHIELDS M.W. : On the problem of achieving adequacy of concurrent programs. Computing Laboratory, University of Newcastle upon Tyne, Tech. Report Series No. 103, 1977. Also in the book: IFIP TC-2 Working Conference on the Formal Description of Programming Concepts, St Andrews, Canada, 1977. North-Holland Pub. Co.

[LC75] LAUER P.E., CAMPBELL R.H. : Formal Semantics for a class of high level primitives for coordinating concurrent processes. Acta Informatica 5, pp 297-332, 1975.

[LS77] LAUER P.E. and SHIELDS M.W. : Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts. Tech. Rep. 117, Computing Laboratory, University of Newcastle-upon-Tyne also Proc. of a workshop on Global Description Methods for Synchronisation in Real Time Applications, AFCET, Paris,NOV. 3-4,1977.

[LS78] LAUER P.E., SHIELDS M.W. : Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts. SIGPLAN Notices, Vol. 13, No. 12, December 1978.

[LS80] LAUER P.E., SHIELDS M.W. : COSY: An Environment for Development and Analysis of Concurrent and Distributed Systems. Proc. of Symposium on Software Engineering Environments, Lahnstein, June 1980 (Ed. H Hunke), North-Holland Publishing Co. 1981.

[LS81] LAUER P.E., SHIELDS M.W. : Interpreted COSY programs: Programming and Verification, Proceedings 2nd International Conference on Distributed Computing Systems, Paris, 8-10 April 1981, IEEE Computer Society Press, ed. E. Gelenbe, 1981.

[LSB79a] LAUER P.E., SHIELDS M.W., BEST E. : The design and certification of asynchronous systems of processes. Advanced Course on Abstract Software Specification, Lyngby, Denmark, January 1979. Lecture Notes in Computer Science 86, Springer Verlag, 1979.

[LSB79b] LAUER P.E., SHIELDS M.W., BEST E. : Formal Theory of the Basic COSY Notation. The Computing Laboratory, University of Newcastle upon Tyne, Tech. Report Series No. 143, November 1979.

[LSC81] LAUER P.E., SHIELDS M.W., COTRONIS J.Y. : Formal Behavioural Specification of Concurrent Systems without Globality Assumptions. Tech. Rep. 162 Computing Laboratory, University of Newcastle-upon-Tyne, also Lecture Notes in Computer Science, Vol 107, Formalisation of Programming Concepts.

[LT78] LAUER P.E., TORRIGIANI P.R. : Toward a System Specification Language Based on Paths and Processes. Tech. Rep. 120, February 78, Computing Laboratory, University of Newcastle-upon-Tyne.

[LTD79] LAUER P.E., TORRIGIANI P.R., DEVILLERS R.E. : The Hyperfast Banker. ASM/55, Computing Laboratory, University of Newcastle-upon-Tyne, January 1979.

[LTD80] LAUER P.E., TORRIGIANI P.R., DEVILLERS R. : A COSY Banker: Specification of Highly Parallel and Distributed Resource Management, Technical Report No 151, Computing Laboratory, University of Newcastle upon Tyne, 1980.

[LTS79] LAUER P.E., TORRIGIANI P.R., SHIELDS M.W. : COSY: A System Specification Language Based on Paths and Processes. Acta Informatica 12, 1979.

[M77] MAZURKIEWICZ A. : Concurrent program schemes and their interpretations. Proc. Aarhus Workshop on Verification of Parallel Processes, June 13-24, Aarhus, Denmark, 1977.

[M80] MILNER R. : A Calculus of Communicating Systems, Lecture Notes in Computer Science No 92, 1980. Springer Verlag.

[M72] MINSKY M. : COMPUTATION: Finite and Infinite Machines. Prentice Hall International, INC, London, 1979.

[OG76] OWICKI S., GRIES D. : An Axiomatic Proof Technique For Parallel Programs. Acta Informatica Vol 6, pp. 319-340, 1976.

[P73] PETRI C.A. : Concepts in Net Theory. Proc. of MFCS, High Tatras, Math. Inst. of the Slovak Academy of Sciences, 1973.

[P75] PETRI C.A. : General Net Theory. Boston Conference on Petri Nets and Related Methods, MIT, 1975.

[P77] PETRI C.A. : Non-Sequential Processes. Technical Report ISF-77-05, GMD, Bonn, 1977.

[P78] PETRI C.A. : Concurrency as a basis of Systems Thinking. GMD Internal Report ISF-78-06, September 1978.

[S73] SALOMAA A. : Formal Languages. ACM Monograph, Academic Press, 1973.

[S79] SHIELDS M.W. : Adequate Path Expressions. Technical Report Number 141, Computing Laboratory, University of Newcastle-upon-Tyne, June 1979.

[S81] SHIELDS M.W. : On the non-sequential behaviour of a class of systems satisfying a generalised free-choice property. Technical Report CRS 92-81, Computer Science Dept., University of Edinburgh.

[SL77] SHIELDS M.W., LAUER P.E. : The equivalence of path expressions and extended semaphore primitives. ASM/43, Computing Laboratory, University of Newcastle-upon-Tyne, November 1977.

[SL78] SHIELDS M.W., LAUER P.E. : On the abstract specification and formal analysis of synchronization properties of concurrent systems. Proc. of Int. Conf. on Mathematical Studies of Information Processing, Aug 23-26, Kyoto, 1978. Lecture Notes in Computer Science 75, Springer Verlag 1979, pp 1-32.

[SL79] SHIELDS M.W., LAUER P.E. : A formal semantics for concurrent systems. Proc. 6th Int. Colloq. for Automata, Languages and Programming, Graz, July 1979, Lecture Notes in Computer Science 71, Springer Verlag, 1979, pp. 569-584.

[SL80a] SHIELDS M.W., LAUER P.E. : Verifying concurrent system specifications in COSY. Proc. 8th Symposium on Mathematical Foundations of Computer Science, Poland, August 1980. Lecture Notes in Computer Science 88, Springer Verlag 1980, pp 576-586.

[SL80b] SHIELDS M.W., LAUER P.E. : Programming and Verifying Concurrent Systems in COSY, Technical Report No 155, Computing Laboratory, University of Newcastle upon Tyne, 1980.

[T78] TORRIGIANI P.R. : Synchronic Aspects of Data Types: Construction of a non-algorithmic solution of the Banker's problem. ECI 78, Information System Methodology, Lecture Notes in Computer Science, Springer-Verlag, 1978.

[TL77] TORRIGIANI P.R., LAUER P.E. : An Object Oriented Notation for Paths and Processes. AICA Annual Conference, Pisa, Vol. 3, pp 349-371(1977).

[ZH80] ZHOU CHAO CHEN, HOARE C.A.R. : Partial Correctness of Communicating Sequential Processes. Proceedings 2nd International Conference on Distributed Computing Systems Paris 8-10 April 1981, IEEE Computer Society Press, ed. E. Gelenbe, 1981.

Appendix A

THE SYNTAX OF PROGRAMS IN THE BASIC COSY NOTATION

A-1 THE SYNTAX OF BASIC COSY PROGRAMS WITH SIMPLE OPERATIONS

BN1. basicprogram=program programbody endprogram
BN2. programbody={path/process}+
BN3. path=path (sequence)* end
BN4. process=process (sequence)* end
BN5. sequence={orelement @;}+
BN6. orelement={starelement @,}+
BN7. starelement=element/element*
BN8. element=operation/(sequence)
BN9. operation=lc-letter{lc-letter/digit/_}*
BN10. lc-letter=a/b/.../z
BN11. digit=0/1/.../9

A-2 THE SYNTAX OF BASIC COSY PROGRAMS WITH SUBSCRIPTED OPERATIONS

BN1. - BN8.
BN9a. operation=simple-op/subscr-op
BN9b. simple-op=lc-letter{lc-letter/digit/_}*
BN9c. subscr-op=uc-letter{uc-letter/digit/_}*({integer @,}+)
BN11.
BN12. uc-letter=A/B/.../Z
BN13. integer={digit}+

A-2.1 Context Sensitive Restrictions

(Brest)

Subscripted operations of the same collective name should have the same number of dimensions.

Appendix B

THE SYNTAX OF MACRO PROGRAMS IN THE GENERAL MACRO NOTATION

MN1. mprogram=program mprogrambody endprogram

MN2. mprogrambody={collectivisor/mpath/mprocess/bodyreplicator}+

MN3. collectivisor=array{simpleardecl/replardecl}+endarray

MN4. simpleardecl={arrayid}+({{iexpr:/}iexpr@,}+)

MN5. replardecl= index_spec[{replardecl/arrayid({iexpr @,}+)}+]

MN6. index_spec=#index:iexpr,iexpr,iexpr

MN7. arrayid=uc-letter{uc-letter/digit/_}*

MN8. bodyreplicator=index_spec[{mpath/mprocess/bodyreplicator}+]

MN9. mpath=path (msequence)* end

MN10. mprocess=process (msequence)* end

MN11. msequence={morelement @;}+

MN12. morelement={gelement @,}+

MN13. gelement={rreplicator}*
{starelement/sreplicator/distributor}
{lreplicator}*

MN14. starelement=element/element*

MN15. element=operation/indexedop/(msequence)

MN16. operation=lc-letter{lc-letter/digit/_}*

MN17. indexedop=arrayid{({iexpr@,}+)/ }

MN18. sreplicator=index_spec[{concseq/imbrseq}]

MN19. distributor={;/,}{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]

MN20. lreplicator=index_spec[{;/,}|{concseq/imbrseq}]

```
MN21. rreplicator=index_spec[{concseq/imbrseq}|{;/,}]

MN22. concseq={morelement;}* concor /{morelement;}+ @

MN23. concor={gelement,}+ @

MN24. imbrseq=imbr_at_seq /{morelement ;}* imbror {; morelement}*

MN25. imbror={gelement ,}* imbrstarel {, gelement}*

MN26. imbrstarel=imbrel/imbrel*

MN27. imbrel=(imbrseq)


MN28. imbr_at_seq=
      {morelement ;}+ {@/at_or1f/at_or1m/at_or1b} {; morelement}*;
              {@/at_or1f/at_or1m/at_or1b} {; morelement}+
     /{morelement ;}+ {@/at_or1f/at_or1m/at_or1b} {; morelement}*;
              {at_or1f/at_or1m}
     /{at_or1m/at_or1b} {; morelement}*;
              {@/at_or1f/at_or1m/at_or1b} {; morelement}+
     /{at_or1m/at_or1b} {; morelement}*; {at_or1f/at_or1m}
     /{morelement;}+ {at_or2fb/at_or2fm/at_or2mm/at_or2mb}{;morelement}+
     /{morelement ;}+ {at_or2fm/at_or2mm}
     /{at_or2mm/at_or2mb} {; morelement}+
     /at_or2mm
     /@ {morelement ;}* {at_or1f/at_or1m}
     /@ {morelement ;}* {@/at_or1f/at_or1m/at_or1b} {;morelement}+
     /{at_or1m/at_or1b} {; morelement}* @
     /{morelement ;}+ {@/at_or1f/at_or1m/at_or1b} {;morelement}* @
     /@ msequence @

MN29. at_or1f=@ {,gelement}+
```

MN30. at_or1m={gelement ,}+ @ {, gelement}+

MN31. at_or1b={gelement ,}+ @

MN32. at_or2fb=@ {, gelement}* , @

MN33. at_or2fm=@ {, gelement}+ , @ {, gelement}+

MN34. at_or2mb={gelement ,}+ @ {,gelement}* , @

MN35. at_or2mm={gelement ,}+ @ {,gelement}* , @ {, gelement}+

MN36. uc-letter=A/.../Z

MN37. lc-letter=a/.../z

MN38. digit=0/.../9

MN39. iexpr={+/-/ } {term @{+/-}}+

MN40. term={factor @{*/DIV/MOD/EXP}}+

MN41. factor=integer/constant/index/funct_desig/(iexpr)

MN42. integer={digit}+

MN43. constant=lc-letter{lc-letter/digit/_}*

MN44. index=lc-letter{lc-letter/digit/_}*

MN45. funct_desig={ABS/FACT/SQUARE}(iexpr)

B-1 CONTEXT SENSITIVE RESTRICTIONS

(MPrest)

Collective names should be declared before any path or process involving any of its subscripted operations.

(Crestl)

the upperbound of the dimensions of the collective names to be greater than or equal to their corresponding implicit or explicit lowerbound.

(Crest2)

Each replicator must specify a non empty range for its index.

(Crest3)

All expressions indexing collective names should yield integers for all the values which the indices they involve take.

(Crest4)

A collectivisor involving nested replicators must be of the form

#kn: inn, fin, incn[...#k1: in1, fi1, incl[Y(h1, h2,..., hn)]...]

where hi for i=1,...,n are expressions involving indices kj for j=1,...,n such that each ki for i=1,...,n must appear in at least one dimension, and an index ki i=1,...,n may only appear together with indices kj for j>i in a single expression and in at most (i-1) expressions with indices kj for j<i.

(Irest1)

Identifiers for replicator indices should be distinct from any identifiers used for simple operations.

(Irest2)

Replicator indices are only defined inside "[ ]" of the replicator with which they are associated. In the scope of a replicator index no other replicator index having the same identifier is permitted.

(BRrest)

The range of the bodyreplicator indices should be non empty.

(Rrest)

inc≠0 and n=(fi-in)//inc+1>0 or t' non empty.

(Rrest2)

The replicators should generate subscripted operations permitted by the collectivisors.

(Drest1)

When a subrange is defined the slices will not be required to contain the same number of sections but at least as many sections as specified by the subrange.

(Drest2)

Inside a k-nested distributor there must only be arrays with at least k dimensions out of which exactly k should be specified as their distributable dimensions.

(Drest3)

incind≠0 and Ns=(fiind-inind)//incind+1≥1

(Drest4)

1≤inind+(j-1)*incind≤Ms for j=1,...,Ns

(Drest5)

The dimension selectors in distributors must have values which are meaningful dimensions of array slices.

(Crest5)

An array identifier may only occur once in collectivisors.

Appendix C

THE SYNTAX OF MACRO PROGRAMS IN THE STRICT MACRO NOTATION

```
MN1.  mprogram=program mprogrambody endprogram

RN1.  mprogrambody={collectivisor/mpath/bodyreplicator}+

MN3.  collectivisor=array {simpleardecl/replardecl} +endarray

MN4.  simpleardecl={arrayid}+({{iexpr:/}iexpr @,}+)

MN5.  replardecl=index_spec[{replardecl/arrayid({iexpr @,}+)}+]

MN6.  index_spec=#index:iexpr,iexpr,iexpr

MN7.  arrayid=upper-case-letter{upper-case-letter/digit/_}*

RN2.  bodyreplicator=index_spec[{mpath/bodyreplicator}+]

MN9.  mpath=path (msequence)* end

RN3.  msequence={seqpart @;}+

RN4.  seqpart=seqmacro/morelement

RN5.  morelement={orpart @,}+

RN6.  orpart=ormacro/starmacro/mstarelement

RN7.  mstarelement=element/element*

RN8.  element=operation/indexedop/(msequence)/elmacro

MN16. operation=lower-case-letter{lower-case-letter/digit/_}*

MN17. indexedop=arrayid{({iexpr @,}+)/ }

RN9.  seqmacro=seqrepl/seqdistr

RN10. seqrepl=index_spec[{seqconcseq/seqimbrseq}]

RN11. seqconcseq={seqpart;}+ {@/seqconcor}

RN12. seqconcor={orpart ,}+ @

RN13. seqdistr=;{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]
                /,{/iexpr}{/#iexpr,iexpr,iexpr}[seqpart{;seqpart}+]
```

```
RN14. seqimbrseq=seqimbr_atout_seq/seqimbr_out_seq

RN15. seqimbr_out_seq={seqpart ;}+ seqimbror {; seqpart}*
                      /{seqpart ;}* seqimbror {; seqpart}+

RN16. seqimbr_atout_seq=
          {seqpart ;}+ {@/at_orlf/at_orlm/at_orlb} {; seqpart}*;
                       {@/at_orlf/at_orlm/at_orlb} {; seqpart}+
         /{seqpart ;}+ {@/at_orlf/at_orlm/at_orlb} {; seqpart}*;
                       {at_orlf/at_orlm}
         /{at_orlm/at_orlb} {; seqpart}*;
                       {@/at_orlf/at_orlm/at_orlb} {; seqpart}+
         /{seqpart;}+ {at_or2fb/at_or2fm/at_or2mm/at_or2mb}{;seqpart}+
         /{seqpart ;}+ {at_or2fm/at_or2mm}
         /{at_or2mm/at_or2mb} {; seqpart}+
         /@ {seqpart ;}* {@/at_orlf/at_orlm/at_orlb} {;seqpart}+
         /{seqpart ;}+ {@/at_orlf/at_orlm/at_orlb} {;seqpart}* @
         /@ msequence @

RN17. seqimbror={orpart ,}* seqimbrstarel {, orpart}*

RN18. seqimbrstarel=seqimbrel/seqimbrel*

RN19. seqimbrel=({seqimbr_atin_seq/seqimbr_in_seq})

RN20. seqimbr_in_seq={seqpart ;}* seqimbror {; seqpart}*

RN21. seqimbr_atin_seq=seqimbr_atout_seq
                       /{at_orlm/at_orlb} {; seqpart}* ;{at_orlf/at_orlm}
                       /at_or2mm
                       /@ {seqpart ;}* {at_orlf/at_orlm}
                       /{at_orlm/at_orlb} {; seqpart}* @

RN22. ormacro=orrepl/ordistr
```

```
RN23. ordistr=,{/iexpr}{/#iexpr,iexpr,iexpr}[morelement]

RN24. orrepl=index_spec[{orconcor/orimbror}]

RN25. orconcor={orpart ,}+@

RN26. orimbror=orimbr_atout_or
              /{orpart ,}+ orimbrstarel {, orpart}*
              /{orpart ,}* orimbrstarel {, orpart}+

RN27. orimbr_atout_or=@ {at_orlf/at_orlm}
                        /{at_orlm/at_orlb} @
                        /at_or2mm
                        /@ morelement @

RN28. orimbrstarel=orimbrel/orimbrel*

RN29. orimbrel=(orimbrseq)

RN30. orimbrseq={seqpart ;}* orimbr_in_or {; seqpart}*
                /orimbr_atin_seq

RN31. orimbr_in_or={orpart ,}* orimbrstarel {, orpart}*

RN32. orimbr_atin_seq=
            orimbr_atout_or
           /{seqpart;}+ {at_or2fb/at_or2fm/at_or2mm/at_or2mb}{;seqpart}+
           /{seqpart ;}+ {at_or2fm/at_or2mm}
           /{at_or2mm/at_or2mb} {; seqpart}+
           /@ {@/at_orlf/at_orlm/at_orlb} {;seqpart}+
           /{seqpart ;}+ {@/at_orlf/at_orlm/at_orlb} @

RN33. at_orlf=@ {,orpart}+

RN34. at_orlm={orpart ,}+ @ {, orpart}+

RN35. at_orlb={orpart ,}+ @

RN36. at_or2fb=@ {, orpart}* , @
```

```
RN37. at_or2fm=@ {, orpart}+ , @ {, orpart}+

RN38. at_or2mb={orpart ,}+ @ {,orpart}* , @

RN39. at_or2mm={orpart,}+ @ {,orpart}* ,@ {,orpart}+

RN40. starmacro=index_spec[(starimbrseq)*]

RN41. starimbrseq=starimbr_at_seq
                  /{seqpart ;}* starimbror {; seqpart}*

RN42. starimbror={orpart ,}* starimbrstarel {, orpart}*

RN43. starimbrstarel=starimbrel/starimbrel*

RN44. starimbrel=(starimbrseq)

RN45. starimbr_at_seq=
    {seqpart ;}+ @ ;{mstarelement ;/ } @ {; seqpart}+
  /{seqpart ;}+
      {starat_orlf_one/starat_orlb_one/starat_lb_many/starat_orlm_b}
        ; @ {; seqpart}+
  /{seqpart ;}+ @
      ; {starat_orlf_one/starat_orlf_many/starat_orlb_one/starat_orlm_f}
        {;seqpart}+
  /{seqpart ;}+ {@/starat_orlb_one/starat_orlb_many} ;
       {starat_orlf_one/starat_orlf_many/starat_orlm_f}
  /{starat_orlb_one/starat_orlb_many/starat_orlm_b} ;
       {@/starat_orlf_one/starat_orlf_many} {; seqpart}+
  /{starat_orlb_one/starat_orlb_many} ;
       {starat_orlf_one/starat_orlf_many/starat_orlm_f}
  /{starat_orlb_one/starat_orlb_many/starat_orlm_b} ;
       {starat_orlf_one/starat_orlf_many}
```

```
/{seqpart ;}+
      {starat_or2fb/starat_or2fm/starat_or2mm/starat_or2mb}
         {;seqpart}+
   /{seqpart ;}+ {starat_or2fm/starat_or2mm}
   /{starat_2mb/starat_or2mm} {; seqpart}
   /starat_or2mm
   /@ {starat_orlf_one/starat_orlf_many/starat_orlm_f}
   /@ {@/starat_orlf_one/starat_orlf_many/starat_orlm_f/starat_orlb_one}
      {; seqpart}+
   /{starat_orlb_one/starat_orlb_many/starat_orlm_b} @
   /{seqpart ;}+
     {@/starat_orlb_one/starat_orlb_many/starat_orlm_b/starat_orlf_one}@
   /@ mstarelement @
RN46. starat_orlf_one=@ , mstarelement
RN47. starat_orlf_many=@ {, orpart}+
RN48. starat_orlb_one=mstarelement , @
RN49. starat_orlb_many={orpart ,}+ @
RN50. starat_orlm_f=mstarelement ,@ {, orpart}+
RN51. starat_orlm_b={orpart ,}+ @, mstarelement
RN52. starat_or2fb=@ {,mstarelement/} @
RN53. starat_or2fm=@ {,mstarelement/} ,@ {, orpart}+
RN54. starat_or2mm={orpart ,}+ @ {,mstarelement/}, @ {, orpart}+
RN55. starat_or2mb={orpart ,}+ @ {,mstarelement/}, @
RN56. elmacro=index_spec[(elimbrseq)]
RN57. elimbrseq=elimbr_at_seq
                /{seqpart ;}* elimbror {; seqpart}*
```

```
RN58. elimbror={orpart ,}* elimbrstarel {, orpart}*

RN59. elimbrstarel=elimbrel/elimbrel*

RN60. elimbrel=(elimbrseq)

RN61. elimbr_at_seq=
    {seqpart ;}+ @ ;{element ;/ } @ {; seqpart}+
   /{seqpart ;}+
      {elat_orlf_one/elat_orlb_one/elat_lb_many/elat_orlm_b}
         ; @ {; seqpart}+
   /{seqpart;}+
      @;{elat_orlf_one/elat_orlf_many/elat_orlb_one/elat_orlm_f}
         {;seqpart}+
   /{seqpart ;}+ {@/elat_orlb_one/elat_orlb_many} ;
      {elat_orlf_one/elat_orlf_many/elat_orlm_f}
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} ;
      {@/elat_orlf_one/elat_orlf_many} {; seqpart}+
   /{elat_orlb_one/elat_orlb_many} ;
      {elat_orlf_one/elat_orlf_many/elat_orlm_f}
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} ;
      {elat_orlf_one/elat_orlf_many}
   /{seqpart ;}+
      {elat_or2fb/elat_or2fm/elat_or2mm/elat_or2mb} {;seqpart}+
   /{seqpart ;}+ {elat_or2fm/elat_or2mm}
   /{elat_2mb/elat_or2mm} {; seqpart}
   /elat_or2mm
   /@ {elat_orlf_one/elat_orlf_many/elat_orlm_f}
```

```
   /@ {@/elat_orlf_one/elat_orlf_many/elat_orlm_f/elat_orlb_one}
      {; seqpart}+
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} @
   /{seqpart ;}+
      {@/elat_orlb_one/elat_orlb_many/elat_orlm_b/elat_orlf_one} @
   /@ element @
RN62. elat_orlf_one=@ , element
RN63. elat_orlf_many=@ {, orpart}+
RN64. elat_orlb_one=element , @
RN65. elat_orlb_many={orpart ,}+ @
RN66. elat_orlm_f=element ,@ {, orpart}+
RN67. elat_orlm_b={orpart ,}+ @, element
RN68. elat_or2fb=@ {,element/} @
RN69. elat_or2fm=@ {,element/} ,@ {, orpart}+
RN70. elat_or2mm={orpart ,}+ @ {,element/}, @ {, orpart}+
RN71. elat_or2mb={orpart ,}+ @ {,element/}, @
MN36. uc-letter=A/.../Z
MN37. lc-letter=a/.../z
MN38. digit=0/.../9
MN39. iexpr={+/-/ } {term @{+/-}}+
MN40. term={factor @{*/DIV/MOD/EXP}}+
MN41. factor=integer/constant/index/funct_desig
MN42. integer={digit}+
MN43. constant=lc-letter{lc-letter/digit/_}*
MN44. index=lc-letter{lc-letter/digit/_}*
MN45. funct_desig={ABS/FACT/SQUARE}(iexpr)
```

Appendix D

THE SYNTAX OF MACRO PROGRAMS IN THE CONSTRAINED MACRO NOTATION

MN1. mprogram=program mprogrambody endprogram

RN1. mprogrambody={collectivisor/mpath/bodyreplicator}+

MN3. collectivisor=array {simpleardecl/replardecl}+ endarray

MN4. simpleardecl={arrayid}+({{iexpr:/}iexpr@,}+)

MN5. replardecl=index_spec[{replardecl/arrayid({iexpr @,}+)}+]

MN6. index_spec=#index:iexpr,iexpr,iexpr

MN7. arrayid=uc-letter{uc-letter/digit/_}*

RN2. bodyreplicator=index_spec[{mpath/bodyreplicator}+]

MN9. mpath=path (msequence)* end

RN3. msequence={seqpart @;}+

RN4. seqpart=seqmacro/morelement

RN5. morelement={orpart @,}+

RN6. orpart=ormacro/starmacro/mstarelement

RN7. mstarelement=element/element*

RN8. element=operation/indexedop/(msequence)/elmacro

MN16. operation=lc-letter{lc-letter/digit/_}*

MN17. indexedop=arrayid{({iexpr @,}+)/ }

RN9. seqmacro=seqrepl/seqdistr

RN10. seqrepl=index_spec[{seqconcseq/seqimbrseq}]

CN1. seqdistr=;{/iexpr}{/#iexpr,iexpr,iexpr}[msequence]

CN2. seqconcseq={seqpart;}+@

```
CN3.  seqimbrseq=seqimbr_at_seq
                 /{seqpart ;}+ seqimbror {; seqpart}*
                 /{seqpart ;}* seqimbror {; seqpart}+
CN4.  seqimbror={orpart,}* seqimbrstarel {, orpart}*
CN5.  seqimbrstarel=seqimbrel/seqimbrel*
CN6.  seqimbrel=(seqimbrseq)
CN7.  seqimbr_at_seq=
          {seqpart;}+ {@/at_orlf} {;seqpart}* ; {@/at_orlb} {;seqpart}+
          /{seqpart ;}+ at_or2fb {; seqpart}+
          /@{seqpart;}* {@/at_orlb} {;seqpart}+
          /{seqpart;}+ {@/at_orlf} {;seqpart}*@
          /@ msequence @
RN22. ormacro=orrepl/ordistr
RN23. ordistr=,{/iexpr}{/#iexpr,iexpr,iexpr}[morelement]
RN24. orrepl=index_spec[{orconcor/orimbror}]
RN25. orconcor={orpart ,}+@
RN26. orimbror=orimbr_atout_or
                 /{orpart ,}+ orimbrstarel {, orpart}*
                 /{orpart ,}* orimbrstarel {, orpart}+
RN27. orimbr_atout_or=@ {at_orlf/at_orlm}
                        /{at_orlm/at_orlb} @
                        /at_or2mm
                        /@ morelement @
RN28. orimbrstarel=orimbrel/orimbrel*
RN29. orimbrel=(orimbrseq)
```

```
RN30. orimbrseq={seqpart ;}* orimbr_in_or {; seqpart}*
                /orimbr_atin_seq

RN31. orimbr_in_or={orpart ,}* orimbrstarel {, orpart}*

RN32. orimbr_atin_seq=
          orimbr_atout_or
         /{seqpart;}+ {at_or2fb/at_or2fm/at_or2mm/at_or2mb}{;seqpart}+
         /{seqpart ;}+ {at_or2fm/at_or2mm}
         /{at_or2mm/at_or2mb} {; seqpart}+
         /@ {@/at_orlf/at_orlm/at_orlb} {;seqpart}+
         /{seqpart ;}+ {@/at_orlf/at_orlm/at_orlb} @

RN33. at_orlf=@ {,orpart}+

RN34. at_orlm={orpart ,}+ @ {, orpart}+

RN35. at_orlb={orpart ,}+ @

RN36. at_or2fb=@ {, orpart}* , @

RN37. at_or2fm=@ {, orpart}+ , @ {, orpart}+

RN38. at_or2mb={orpart ,}+ @ {,orpart}* , @

RN39. at_or2mm={orpart,}+ @ {,orpart}* ,@ {,orpart}+

RN40. starmacro=index_spec[(starimbrseq)*]

RN41. starimbrseq=starimbr_at_seq
                 /{seqpart ;}* starimbror {; seqpart}*

RN42. starimbror={orpart ,}* starimbrstarel {, orpart}*

RN43. starimbrstarel=starimbrel/starimbrel*

RN44. starimbrel=(starimbrseq)

RN45. starimbr_at_seq=
    {seqpart ;}+ @ ;{mstarelement ;/ } @ {; seqpart}+
```

```
/{seqpart ;}+
      {starat_orlf_one/starat_orlb_one/starat_lb_many/starat_orlm_b}
         ; @ {; seqpart}+
   /{seqpart ;}+ @
      ; {starat_orlf_one/starat_orlf_many/starat_orlb_one/starat_orlm_f}
         {;seqpart}+
   /{seqpart ;}+ {@/starat_orlb_one/starat_orlb_many} ;
       {starat_orlf_one/starat_orlf_many/starat_orlm_f}
   /{starat_orlb_one/starat_orlb_many/starat_orlm_b} ;
       {@/starat_orlf_one/starat_orlf_many} {; seqpart}+
   /{starat_orlb_one/starat_orlb_many} ;
       {starat_orlf_one/starat_orlf_many/starat_orlm_f}
   /{starat_orlb_one/starat_orlb_many/starat_orlm_b} ;
       {starat_orlf_one/starat_orlf_many}
   /{seqpart ;}+
      {starat_or2fb/starat_or2fm/starat_or2mm/starat_or2mb}
         {;seqpart}+
   /{seqpart ;}+ {starat_or2fm/starat_or2mm}
   /{starat_2mb/starat_or2mm} {; seqpart}
   /starat_or2mm
   /@ {starat_orlf_one/starat_orlf_many/starat_orlm_f}
   /@ {@/starat_orlf_one/starat_orlf_many/starat_orlm_f/starat_orlb_one}
      {; seqpart}+
   /{starat_orlb_one/starat_orlb_many/starat_orlm_b} @
   /{seqpart ;}+
     {@/starat_orlb_one/starat_orlb_many/starat_orlm_b/starat_orlf_one}@
```

```
   /@ mstarelement @

RN46. starat_orlf_one=@ , mstarelement

RN47. starat_orlf_many=@ {, orpart}+

RN48. starat_orlb_one=mstarelement , @

RN49. starat_orlb_many={orpart ,}+ @

RN50. starat_orlm_f=mstarelement ,@ {, orpart}+

RN51. starat_orlm_b={orpart ,}+ @, mstarelement

RN52. starat_or2fb=@ {,mstarelement/} , @

RN53. starat_or2fm=@ {,mstarelement/} ,@ {, orpart}+

RN54. starat_or2mm={orpart ,}+ @ {,mstarelement/}, @ {, orpart}+

RN55. starat_or2mb={orpart ,}+ @ {,mstarelement/}, @

RN56. elmacro=index_spec[(elimbrel)]

RN57. elimbrseq=elimbr_at_seq
                /{seqpart ;}* elimbror {; seqpart}*

RN58. elimbror={orpart ,}* elimbrstarel {, orpart}*

RN59. elimbrstarel=elimbrel/elimbrel*

RN60. elimbrel=(elimbrseq)

RN61. elimbr_at_seq=
     {seqpart ;}+ @ ;{element ;/ } @ {; seqpart}+
    /{seqpart ;}+
       {elat_orlf_one/elat_orlb_one/elat_lb_many/elat_orlm_b}
          ; @ {; seqpart}+
    /{seqpart;}+
       @;{elat_orlf_one/elat_orlf_many/elat_orlb_one/elat_orlm_f}
          {;seqpart}+
```

```
/{seqpart ;}+ {@/elat_orlb_one/elat_orlb_many} ;
       {elat_orlf_one/elat_orlf_many/elat_orlm_f}
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} ;
      {@/elat_orlf_one/elat_orlf_many} {; seqpart}+
   /{elat_orlb_one/elat_orlb_many} ;
      {elat_orlf_one/elat_orlf_many/elat_orlm_f}
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} ;
      {elat_orlf_one/elat_orlf_many}
   /{seqpart ;}+
      {elat_or2fb/elat_or2fm/elat_or2mm/elat_or2mb}
         {;seqpart}+
   /{seqpart ;}+ {elat_or2fm/elat_or2mm}
   /{elat_2mb/elat_or2mm} {; seqpart}
   /elat_or2mm
   /@ {elat_orlf_one/elat_orlf_many/elat_orlm_f}
   /@ {@/elat_orlf_one/elat_orlf_many/elat_orlm_f/elat_orlb_one}
      {; seqpart}+
   /{elat_orlb_one/elat_orlb_many/elat_orlm_b} @
   /{seqpart ;}+
      {@/elat_orlb_one/elat_orlb_many/elat_orlm_b/elat_orlf_one} @
   /@ element @
RN62. elat_orlf_one=@ , element
RN63. elat_orlf_many=@ {, orpart}+
RN64. elat_orlb_one=element , @
RN65. elat_orlb_many={orpart ,}+ @
RN66. elat_orlm_f=element ,@ {, orpart}+
```

```
RN67. elat_or1m_b={orpart ,}+ @, element

RN68. elat_or2fb=@ {,element/} , @

RN69. elat_or2fm=@ {,element/} ,@ {, orpart}+

RN70. elat_or2mm={orpart ,}+ @ {,element/}, @ {, orpart}+

RN71. elat_or2mb={orpart ,}+ @ {,element/}, @

MN36. uc-letter=A/.../Z

MN37. lc-letter=a/.../z

MN38. digit=0/.../9

MN39. iexpr={+/-/ } {term @{+/-}}+

MN40. term={factor @{*/DIV/MOD/EXP}}+

MN41. factor=integer/constant/index/funct_desig

MN42. integer={digit}+

MN43. constant=lc-letter{lc-letter/digit/_}*

MN44. index=lc-letter{lc-letter/digit/_}*

MN45. funct_desig={ABS/FACT/SQUARE}(iexpr)
```