Software specification languages based on regular expressions

Robert Marti

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Alan C. Shaw

SOFTWARE SPECIFICATION LANGUAGES BASED ON REGULAR EXPRESSIONS

JUNE 1979 31

June, 1979

SOFTWARE SPECIFICATION LANGUAGES BASED ON REGULAR EXPRESSIONS

Alan C. Shaw *)
Department of Computer Science FR-35
University of Washington
Seattle, WA 98195

(presented at the Software Development Tools Workshop Conference, Pingree Park, Colorado, May 21-25, 1979)

Abstract

The behavior of a software system can be modelled in terms of sequences of events, flows, or operations that may occur during execution. To support this approach, a number of non-procedural description languages based on regular expressions have been proposed. These include event expressions, flow expressions, and the many variations of path expressions. The purpose of this paper is to survey and assess these notations, including some of my work in the area, and to suggest some future directions.

*) on leave 1978-9 at the Institut für Informatik, ETH, Zürich

Contents

1. Introduction

1.1 Regular expressions (RE's) have been used as a design and modelling tool for a broad variety of computer system elements, including sequential circuits, lexical analyzers, communication system protocols, command languages, and programs. Since the early 1970's, the RE notation has been adapted, embedded, and/or extended to describe some of the more difficult aspects of sequential and concurrent software, especially the latter. The result is a new set of software specification languages. The purpose of this paper is to survey and assess the developments in RE-based description schemes, including some of my work in the area, and to suggest some future directions.

1.2 Historically and logically, the expression notations can be divided into two classes. One class originated in the research of Riddle, first reported in 1972 [39], in defining and applying message transfer expressions (MTE's), later called event expressions (EE's) [47]. The EE notation is a pure extension of RE's with interleaving operators to model concurrency and with synchronization symbols; the language denoted by an EE is interpreted as a set of event sequences characterizing the behavior of some part of a system. Flow expressions (FE's), the language I devised starting in 1975 [54,55], is similar to EE's but with a different synchronization method. The other class is the path expression (PE) notations introduced by Campbell and Habermann in 1974 [13] and further developed by them, Lauer, and others. Most of the work in this class is aimed specifically at specifying constraints, given by PE's, on the order of execution of operations in encapsulated data or resource objects. While PE's also consist primarily of RE's, they have more of a programming "flavor" than the EE and FE schemes, and do not use explicit synchronization symbols.

Despite differences in motivation, syntax, and interpretation of symbols in expressions, both classes of description methods have many similarities. For example, any of the notations could be viewed as representing flows, paths, or events, and some formal relations between the schemes have been established. Most, but not all, of the applications may be described with any of the methods, the choice being a matter of taste and convenience.

1.3 In the next three sections, I review the RE formalism and its applications in software description; present and compare EE's, FE's, the various reported forms of PE's, and some related proposals; and describe applications of each. The final section contains a general assessment and some ideas for future work. A bibliography and index of symbols and abbreviations are included at the end of the paper. The presentation is informal and the reader is referred to the literature for more precise definitionsand formal results.

There is a natural connection and overlap between non-procedural description techniques, such as the ones discussed here, on the one hand, and procedural methods, analysis and verification tools, and implementation methods, on the other side. Thus, one will find RE ideas being used in all of these areas. I will emphasize the purely descriptive aspect of the notations, but will also on occasion mention these other applications since they follow directly from the former.

2. Regular Expressions and Languages

The underlying idea behind the use of RE's is that the permissible behavior or the constraints of a system can be described by a language, each sentence of which is interpreted as a permissible "trace", "path", "flow", or sequence of "events". The basic symbols or atoms of the language are viewed as indivisible or atomic behavioral descriptors.

2.1 RE's are formed from the basic symbols and operators for
1) concatenation, indicated by juxtaposition of expressions,
2) union or selection, denoted by ∪, and
3) indefinite repetition, represented by the Kleene star * (e.g.[35]).
Thus a b means a followed by b, a ∪ b means either a or b, and a* means zero or more repetitions of a.

Example 1:

Let Insert and Remove be atoms. Then the RE's

Insert , Insert Remove , Insert $\cup$ Remove , and Insert*

describe the languages (sets of strings)

{Insert},{Insert Remove},{Insert,Remove},and{λ,Insert, Insert Insert,Insert Insert Insert,...}, respectively,

where λ is the symbol for the null or empty string.

Parenthesis are used for grouping, so that quite elaborate expressions may be constructed.

Example 2:

The permissible operations on a particular data base may be described by the RE

Open (Read ((Update Write) $\cup$ Print))* Close

which represents the sequences (using O for Open, R for Read, U for Update, W for Write, P for Print, and C for Close):

{O C,O R U W C,O R P C,O R U W R P C,O R U W R U W C,...}.

The intended interpretation is that an Open operation (command, message,...) is followed by zero or more (Read Update Write)'s and/or (Read Print)'s, and finally terminated by a Close.

2.2 In addition to being a simple and convenient notation, RE's have a well-developed theory and their languages can be implemented, i.e. recognized, by relatively simple machines - finite automata or finite state machines. Figure 1 presents a state diagram for recognizing the language of the RE given in Example 2; the circles represent machine states and a labelled edge is a path to the next state that is entered if the next input symbol is equal to the label. The significance for software design is that RE descriptions can be readily used for simulation, analysis, and implementation.

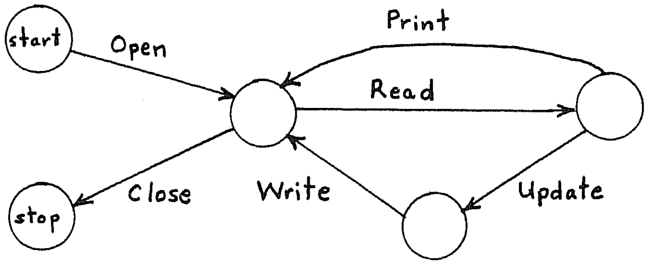


Figure 1

State Diagram For: Open (Read ((Update Write) ∪ Print))* Close

2.3 The basic symbols used in the RE's may stand for a variety of entities, depending on the particular aspect being modelled. For example, the symbols of the RE (a (b ∪ c))* may denote any of the following:

1) message transmissions. a, b, and c are types of messages, such as an end-of-file message, a resource request or release message, or a job log-in message. The RE states that a message of type a must or may be followed by either one of type b or one of type c, and that transmissions of a must occur between b and c.
2) procedure or operation executions. a, b, and c are procedures. The RE expresses the constraints that the execution of a must precede that of b or c and that b and c cannot be executed in sequence without an intervening a.
3) control points. The symbols identify program control locations, such as statement or module labels. In this context, the RE says that control passes from point a to either b or c repeatedly.
4) input language symbols. The RE may describe some part of the user (interactive) input that drives the software. Here a, b, and c could be input lexical items; for example, the RE could be (MOVE (BOX ∪ TEXT))*.
5) resource flows. The symbols name the particular system component, such as a process or a user,that has been allocated a specific

resource, for example, a processor, tape unit, file, or table. The RE states that either component b or c may have the resource after it passes through a, and that a must have it between allocations to b and c.

The above list is not exhaustive but it covers most of the reported software description applications where RE's are used in their pure form. Hierarchical or structured descriptions are easily produced by assigning names to expressions.

3. Event and Flow Expressions

3.1 The Shuffle Operators

Both the EE and FE works rely on an interleaved model to describe concurrencies in systems. The intuitive idea is that the effect of the concurrent execution of two components, say p_1 and p_2, is the same as that obtained by interleaving or shuffling the execution history of p_1 with that of p_2. To describe this interleaving, the RE notation was extended with a shuffle operator and its closure, here denoted $\odot$ and $^{\circledast}$, respectively.

3.1.1 The expression ($S_1 \odot S_2$) represents the shuffle of all elements of S_1 with those of S_2.*) $S^{\circledast}$ means zero or more shuffles of S; it can be used to specify an unbounded or variable degree of concurrency.

Example 3:

1) Read $\odot$ Write describes the language {Read Write, Write Read}.
2) (a b) $\odot$ (c d) produces the set of strings {a b c d, a c b d, a c d b, c a d b, c d a b, c a b d}.
3) $(a\ b\ c)^{\circledast}$ stands for $\lambda \cup (a\ b\ c) \cup ((a\ b\ c) \odot (a\ b\ c)) \cup ((a\ b\ c) \odot (a\ b\ c) \odot (a\ b\ c)) \cup \ldots$

*) Precise, formal definitions of these operators appear, for example, in [47] and [55]. It is hoped that the examples will clarify any problems related to my deliberately informal definitions.

I will use the term shuffle expression (SE) to mean an RE extended with the $\odot$ and $\circledast$ operators. (Aside on history and notation. A shuffle operator similar to $\odot$ was earlier defined in [17] and also appeared in [18]. Riddle uses † and Δ, instead of my $\odot$ and $\circledast$, in his EE work which precedes mine. The main argument for $\odot$ and $\circledast$ is that because they are meant to be interleaved/ concurrent analogs of the sequential concatenation and * operators, respectively, it is convenient to have them similar in appearance.)

Example 4:

Let Read$(r_1,\ldots,r_n)$ denote an SE describing the elementary activities involved in reading a data base, where $r_1,\ldots,r_n$ are the atomic operations of the read. Similarly, let Write$(w_1,\ldots,w_m)$ stand for an SE for the flows through the data base write operations, where $w_1,\ldots,w_m$ are the elementary write activities. Then, the standard reader-writer problem constraints, stating that Read's may proceed concurrently, Write's must proceed sequentially, and a Write may not overlap a Read, can be expressed by the SE:

$$(\ \mathrm{Read}(r_1,\ldots,r_n)^{\circledast} \cup \mathrm{Write}(w_1,\ldots,w_m)\ )^*$$

Example 5:

Consider the SE: $(\mathrm{Put\ Get})^{\circledast}$

The language described is

L = {λ, Put Get, Put Get Put Get, Put Put Get Get, Put Get Put Get Put Get, Put Put Get Get Put Get, Put Put Get Put Get Get,...}.

L has the interesting property that for any string $\alpha = \alpha_1\alpha_2$ in L, where α_1 is any initial substring of α, the number of Put's in α_1 is always greater or equal to the number of Get's in α_1. This SE can model the operation constraints on a data type where retrieval of a data element (Get) is only possible if it can be matched with a preceding insertion (Put), such as an unbounded queue (Put = Insert, Get = Remove), an unbounded stack (Put = Push, Get = Pop), or a general semaphore initialized to zero(Put = V, Get = P).

3.1.2 Theoretically, RE's augmented by only the $\odot$ operator are no more powerful than RE's [17,47]; i.e. the resulting languages can still be described by RE's. However, from a practical viewpoint, the recognizing automata are much more complex [36]. The further addition of ⊛ does increase the expressive power beyond that of RE's though [55]; for example, the language of Example 5 cannot be described by an RE. The exponential parsing algorithm in [56] provides some insight into the practical complexity of parsing general SE's.

3.1.3 The interleaved model of concurrency has attracted some criticism because the "events" comprising concurrent activities become ordered under the shuffling operations whereas they may be truly parallel and asynchronous if, for example, there is no global clock in a system; it has also been argued that shuffling introduces unnecessary details into descriptions compared with a notation based on <u>partial orders</u> (e.g. [19]). I discuss this point further in Section 5. It should be noted, however, that true simultaneity can be handled in SE's by bracketting the relevant entities with start and finish symbols and omitting the rest of the activities [47]; for example, if $start_i$ and $finish_i$ denote the beginning and end of activity i, the SE

$$(start_1\ finish_1)^* \odot (start_2\ finish_2)^*$$

indicates that activity 1 is sequenced, activity 2 is sequenced, and activities 1 and 2 may proceed in parallel.

3.1.4 One unanticipated product of the interleaving research was the discovery of some new and interesting applications in command language description. While a language design tool is not the same as a software design tool, it is nevertheless the case that the design of language processors is strongly influenced by the language specification technique used. Welter first suggested that the shuffle ⊙ may be useful as an extension to BNF for command languages [61]. Applications of FE's in textual and graphics command language specification were demonstrated in [55,56] ; some required the full power of SE's (⊙ and ⊛), as well as the synchronization facility of FE's.

RE's and finite state machines are the most commonly used formal schemes for defining command languages and their parsers. However, RE's are not convenient and/or adequate for some of the desired properties of interactive textual and graphics languages. Some of these properties that can be handled by SE's and FE's are discussed below:

1) It is often immaterial in which order a command and its parameters are entered, and at different times during an interactive session, different orderings may be convenient.

<u>Example 6:</u>

To take a simple case, computing the sum of two numbers x and y could be specified in at least 3 ways:

x + y , + x y , and x y +

Depending on the ways in which x, y, and + can be designated (e.g. pointing on a display, rotating a dial, typing, ...), all three may be desired. These options are described by an SE:

$$\langle + \rangle \circledcirc \langle \text{number} \rangle \circledcirc \langle \text{number} \rangle$$

2) Several different command languages may be active at the same time, especially in graphics applications. At the extreme, each textual or graphical object may have its own language (e.g. [25]).

<u>Example 7:</u>

A text editing and placement language T is often used in conjunction with a drawing language D. (Let D define the operations for drawing and manipulating a box, and T be the means for specifying the text inside a box.) If $T_1, T_2, \ldots, T_n$ are the possible commands of T and $D_1, D_2, \ldots, D_m$ are those of D, then the FE:

$$(T \circledcirc D)^* = ((T_1 \cup T_2 \ldots \cup T_n)^* \circledcirc (D_1 \cup D_2 \ldots \cup D_m)^*)^*$$

is a reasonable description that maintains the separation of the languages yet indicates that the commands may be interleaved.

3) In a manner similar to 2), it is occasionally convenient to interleave several commands or sequences of commands from the <u>same</u> language. Several instantiations of the same command sequence may also be desirable.

Example 8:

Consider the file processing FE of Example 2 as a command language:

Open (Read ((Update Write) $\cup$ Print))* Close .

If one wishes to view and manipulate several files concurrently, the command language would be

(Open (Read ((Update Write) $\cup$ Print))* Close)$^{\circledast}$

This FE reflects, for example, part of a program design system, where each procedure or module is a separate file and several partially defined procedures may be displayed and edited at the same time. Of course, each instantiation resulting from $\circledast$ must be uniquely identified at some point.

An application related to the above that I am developing is the description of interactive document generation systems. Here, the "abstract syntax" of a document - its structure and components - has been defined as a combination of ordered and unordered sets of document objects, each of which can be similarly decomposed. I am using SE's and FE's to describe this abstract syntax.

3.2 Event Expressions

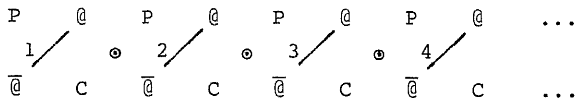
EE's were originally invented to specify the message transmission behavior among a set of interacting processes; for this application, the notation was called message transfer expressions (MTE's) and the basic symbols denote message types [39]. However, the scheme is completely general and can be used in other areas. The name "event expression" reflects this broader view. The basic symbols of EE's denote events - a general term for entities that may be partially or totally ordered by time (e.g. [19,29,38]). Thus, any of the symbol interpretations mentioned at the end of Section 2 are "events".

3.2.1 EE's are SE's with the addition of a synchronization mechanism. The idea is to impose some restrictions on shuffling so that in an interleave such as $(S_1 \circledcirc S_2)$ only a subset of the possible shuffles are in the language; the objective is to describe critical sections, message passing, and other synchronization constraints. .

A point "in time" is marked in an EE with one of the special synchronization symbol pairs $@, \overline{@}, @_1, \overline{@}_1, \ldots$

e.g. $S_1 = (\ P\ @\ )^*$ $S_2 = (\ \overline{@}\ C\ )^*$

The symbols are interpreted so that each @ must be matched with its "inverse" $\overline{@}$ and that the time points marked by each $@\ \overline{@}$ pair are forced to be identical. Taking the shuffle of S_1 and S_2 above, $S_1 \odot S_2$ gives


P @ P @ P @ P @ ...
1 ⊙ 2 ⊙ 3 ⊙ 4
$\overline{@}$ C $\overline{@}$ C $\overline{@}$ C $\overline{@}$ C ...

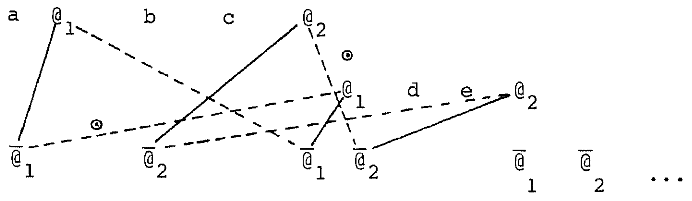
The matched pairs at points 1,2,3,... cause the shuffling to be restricted to $P\ (\ P \odot C\ )^*\ C$.

Critical section locking is specified by surrounding an expression with synchronization symbols for entry and exit.

<u>Example 9:</u>

$(\ a\ @_1\ b\ c\ @_2\ ) \odot (\ @_1\ d\ e\ @_2\ ) \odot (\ \overline{@}_1\ \overline{@}_2\ )^*$

produces matching @ symbols in two possible ways (solid and dotted lines).


a $@_1$ b c $@_2$
⊙
$@_1$ d e $@_2$
⊙
$\overline{@}_1$ $\overline{@}_2$ $\overline{@}_1$ $\overline{@}_2$ $\overline{@}_1$ $\overline{@}_2$ ...

The EE describes the strings

{a b c d e, a d e b c , d a e b c , d e a b c}.

bc and de are thus indivisible with respect to one another, and only 4 of the possible 10 shuffles of (a b c) and (d e) are in the language.

Example 10:

Let the events comprising a data base read, update, write, and print be defined by the EE's:

Read = startread doread endread

Update = startupdate doupdate endupdate

Write = startwrite dowrite endwrite

Print = startprint doprint endprint

The EE:

$$(@_1 \text{ Read } @_2 \; (\; (\text{ Update } @_1 \text{ Write } @_2 \;) \cup \text{ Print }) \;)^{\circledast} \circledcirc \; (\; \overline{@}_1 \; \overline{@}_2 \;)^*$$

describes the behaviors or constraints:

1) Any number of (Read Update Write)'s and (Read Print)'s may be in execution concurrently ($\circledast$).
2) Read's and Write's are mutually exclusive; neither can overlap other Read's or Write's ($@_1, @_2, \overline{@}_1, \overline{@}_2$).

3.2.2 The synchronization method is quite elegant and its meaning can be expressed concisely: Treat the EE first as an ordinary SE and generate the language. Apply the transformations

$$@_i \; \overline{@}_i \rightarrow \lambda \quad \text{and} \quad \overline{@}_i \; @_i \rightarrow \lambda$$

for all synchronization symbols to each sentence repeatedly; this cancels out matched pairs. The language described by the EE is just those resulting sentences that have no remaining (uncancelled) synchronization symbols.

Example 11:

Treated as an SE, the expression $(\; a \; @ \;)^{\circledast} (\; \overline{@} \; b \;)^{\circledast}$ describes the language:

$$\{\lambda,\ a\ @,\ \overline{@}\ b,\ a\ \underline{@\ \overline{@}}\ b,\ a\ @\ a\ @, a\ a\ @\ @,\ \overline{@}\ b\ \overline{@}\ b,\ \overline{@}\ \overline{@}\ b\ b,$$
$$a\ \underline{@\ \overline{@}}\ b\ \overline{@}\ b, \ldots, a\ a\ \underline{@\ \underline{@\ \overline{@}}\ \overline{@}}\ b\ b, \ldots, a\ a\ a\ \underline{@\ \underline{@\ \underline{@\ \overline{@}}\ \overline{@}}\ \overline{@}}\ b\ b\ b\ , \ldots$$

The matched pairs that may be cancelled in the next step are underlined. After cancelling symbols, the resulting set is

$$\{\lambda, a\ @, \overline{@}\ b, a\ b, a\ @\ a\ @, a\ a\ @\ @, \overline{@}\ b\ \overline{@}\ b, \overline{@}\ \overline{@}\ b\ b, a\ b\ \overline{@}\ b, \ldots, a\ a\ b\ b, \ldots, a\ a\ a\ b\ b\ b, \ldots\}.$$

The elements with synchronization symbols still remaining are removed to give the final language, the one described by the EE:

$$\{a^n\ b^n : n \geq 0\}.$$

Example 12:

Consider a producer-consumer problem where a cyclic producer process fills a single slot buffer and a consumer process empties the buffer.

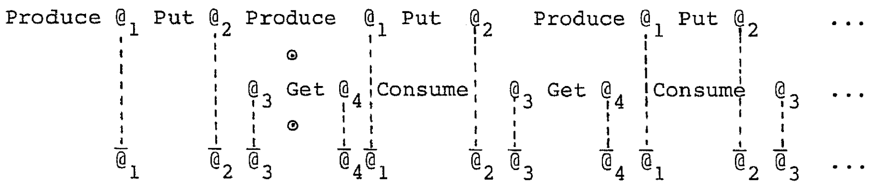
ProducerProcess = (Produce Put)*

ConsumerProcess = (Get Consume)*

We would like an EE, that on the one hand, describes the possible concurrencies among the events comprising Produce, Consume, Get, and Put, and that, at the same time, restricts the Put/Get's to the behavior (Put Get)* defining the allowable operation sequences on a one slot buffer. This behavior is specified by the EE

$$(\text{ Produce } @_1 \text{ Put } @_2\ )^* \odot (\ @_3 \text{ Get } @_4 \text{ Consume })^* \odot (\ \overline{@}_1\ \overline{@}_2\ \overline{@}_3\ \overline{@}_4\ )^*$$

The interactions among the three subexpressions are illustrated below:


Produce @1 Put @2 Produce @1 Put @2 Produce @1 Put @2 ...
⊙
@3 Get @4 Consume @3 Get @4 Consume @3 ...
⊙
@1 @2 @3 @4 @1 @2 @3 @4 @1 @2 @3 ...

The synchronizing expression $(\ \overline{@}_1\ \overline{@}_2\ \overline{@}_3\ \overline{@}_4\ )^*$ ensures the (Put Get)* constraint.

3.2.3 The EE scheme is deceptively powerful. In fact, it is descriptively universal; any language that can be recognized by a Turing machine or a computer with unbounded storage or that can be generated by an unrestricted (type 0) grammar can be described by an EE [36,47]. However, if the $\circledast$ operator is eliminated from EE's, the resulting notation is equivalent only to RE's [47]. While theoretically sufficient for any descriptive task, EE's are sometimes inconvenient when compared with other methods (and vice versa) on particular problems; specific comparisons are made in subsequent sections.

An elaborate set of axioms and rules of inference for manipulating EE's have been devised [39,47]. These are of potential value in proving equivalence of descriptions and deriving performance measures from EE's.

3.2.4 One major application of EE's has been in Riddle's software design methodology where EE's are the medium for behavioral specification and analysis. As outlined in [47], a software system with concurrently operating subsystems is modelled by a "Program Process Modelling Scheme" (PPMS) resembling a programming language but concerned only with message generation and transmission activities. There is a procedure for translating a set of programs (processes) given in the PPMS modelling language into an MTE that specifies the message paths in the system. Related work is reported in [62] where a variant called constrained expressions can be derived from a dynamic process modelling scheme (DPMS).

Another application is the design notation DDN that is part of an interactive software design system under development [46,48,49,50,51]. In DDN, internal system events are defined in terms of systems state transitions, and the desired behavior is specified by EE's.

The PPMS model and derived MTE's have also been studied for performance prediction and analysis [53]. Message generation times and conditional branching probabilities associated with PPMS statements are mapped into time parameters and selection probabilities in the associated MTE's. In principle, the MTE's are then "evaluated" to produce performance measures such as the distribution, expected value, and variance of completion times. As in many of the RE-based applications presented in the paper, this approach appears promising but requires more development and experiences before it can be used as a practical tool.

3.3 Flow Expressions

FE's were developed to describe the flows of computer system entities, such as resources, messages, jobs, commands, and control, through sequential and concurrent software components such as programs, procedures, processes, monitors, data types, and modules. The basic symbols of FE's are interpreted as atomic activities rather than events as in EE's, but this distinction may be academic since either notation may, in principle, be used for the same description task. However, the extensions to SE's are different and result in differences in specification convenience.

3.3.1 While the * operator is retained in FE's to express indefinite and usually finite repetition, a new cyclic operator ∞ is used to characterize the infinite repetitions that appear in cyclic processes and languages with non-terminating sentences [55,56].

Example 13:

The producer-consumer example given in the last section might be more accurately described (excluding synchronization):

$$(\text{Produce Put})^\infty \odot (\text{Get Consume})^\infty$$

The ∞ operator is primarily of descriptive value and appears to be difficult to handle analytically.

3.3.2 The synchronization scheme in FE's has a more direct programming connection than that in EE's, and is the major distinguishing difference between the notations. An FE can be locked for shuffling by bracketing it with lock symbols [,] , ${}_1[$, $]_1$, ${}_2[$, $]_2$, ... Flows inside the brackets are then treated as indivisible when interleaving other flows with the same lock brackets; e.g. ([a] $\odot$ [b c]) describes the language {a b c , b c a}.

Example 14:

1) The EE of Example 9 has the equivalent FE:

$(a\ [\ b\ c\ ]\ )\ \odot\ (\ [\ d\ e\ ]^{\bullet}\ )$

b c and d e are "critical sections".

2) Example 10 may be rewritten as an FE:

$(\ [\ \text{Read}\ ]\ (\ (\ \text{Update}\ [\ \text{Write}\ ]\ )\ \cup\ \text{Print}\ )\ )^{\circledast}$

Example 15:

A high level (gross) description of user job flows in an operating system may be:

$(\ {}_1[\ \text{InputSpoolJob}\ ]_1\ \text{RunJob}\ {}_2[\ \text{OutputSpoolJob}\ ]_2\ )^{\circledast}$

Any number of jobs may be executing concurrently, only one InputSpoolJob and one OutputSpoolJob may proceed at a time but they may overlap each other, and RunJob's can proceed in parallel.

The second and more complex FE synchronization mechanism is a wait and signal facility similar to binary semaphores. The symbols $\sigma, \omega, \sigma_1, \omega_1, \ldots$ are introduced into FE's, where σ_i is the analog of a "send signal i" (Dijkstra V) and ω_i corresponds approximately to a "wait for signal i" (Dijkstra P). A string is in the language described by an FE only if every ω_i can be matched (cancelled) with a preceding closest σ_i. For example, treating $(\ a\ \omega\ b\ )\ \odot\ (\ d\ \sigma\ )$ as an SE, yields the strings:

$$a\ \omega\ b\ d\ \sigma \qquad a\ \omega\ d\ b\ \sigma \qquad a\ d\ \omega\ b\ \sigma \qquad d\ a\ \omega\ b\ \sigma \qquad d\ a\ \omega\ \sigma\ b$$

$$\underset{1}{\underline{d\ a\ \sigma\ \omega\ b}} \qquad \underset{2}{\underline{d\ \sigma\ a\ \omega\ b}} \qquad \underset{3}{\underline{a\ d\ \sigma\ \omega\ b}} \qquad a\ \omega\ d\ \sigma\ b \qquad a\ d\ \omega\ \sigma\ b$$

Only the underlined strings have ω matched with a preceding σ. The FE then just describes the language $\{d\ a\ b\ ,\ a\ d\ b\}$. The synchronization defined by this example is informally given by the following diagram:

$$\begin{tikzcd}
a & \omega & b \\
\odot & & \\
d & \sigma \arrow[uu, dashed] &
\end{tikzcd}$$

Locks are definable using the σ/ω facility; for example ([a] ⊙ [b c]) is equivalent to σ ((ω a σ) ⊙ (ω b c σ)) . However, the locking appears so often in descriptions that it is convenient to have the separate [,] notation, both for brevity and to specifically distinguish this use. Locks are also useful in the command language application (Section 3.1.4), when one wants to selectively prohibit shuffling.

3.3.3 In EE's, the paired symbols are symmetric - both @ $\overline{@}$ and $\overline{@}$ @ are legitimate, and the synchronization symbols must be adjacent (or eventually adjacent after cancellation of intervening symbols). By contrast, the FE scheme is unsymmetric - an ω preceding a σ does not lead to a legitimate sentence, and the symbols need not be adjacent; a σ is capable of "action at a distance" so that a σ b c ω d yields the string a b c d . In addition, unconsumed σ's are lost but do not effect the resulting sentence while unmatched ω's, like uncancelled @'s or $\overline{@}$'s, do not result in sentences.

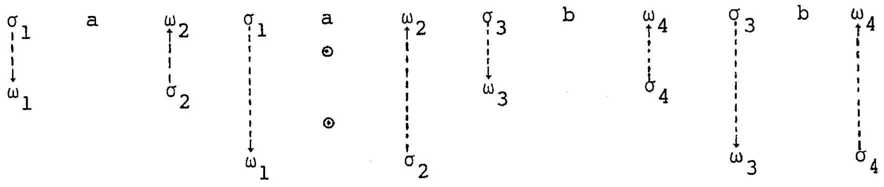
In the critical section (locking) case, these differences lead to easier FE descriptions, that is, fewer symbols are used. The opposite is true for describing some other strings as shown by the next example.

<u>Example 16:</u>

An FE for the language $\{a^n b^n : n \geq 0\}$ (Example 11) is

$$((\sigma_1 \; a \; \omega_2)^* \; (\sigma_3 \; b \; \omega_4 \;)^*) \odot (\; \omega_1 \; \sigma_2 \; \omega_3 \; \sigma_4 \;)^{\circledast}$$

The equivalent EE (a@)⊛($\overline{@}$b)⊛ is much simpler. The following diagram shows how aabb is generated from the FE:



Example 17:

The producer-consumer example (Example 12) with cyclic processes and synchronization is given by the relatively simple FE:

$$\sigma_1 \; (\; (\text{ Produce } \omega_1 \text{ Put } \sigma_2 \;)^{\infty} \odot (\; \omega_2 \text{ Get } \sigma_1 \text{ Consume })^{\infty} \;)$$

Example 18:

A reader-writer problem (Example 4) is treated in more detail. Let a read operation be decomposed into two FE's, one designating a read request and the second denoting the actual read:

Read = ReadRequest ActualRead.

Similarly, let Write = WriteRequest ActualWrite.

Example 4 can then be restated as:

$$(\; (\text{ ReadRequest ActualRead })^{\circledast} \cup (\text{ WriteRequest ActualWrite }) \;)^{*}$$

In these terms, the FE may be overly restrictive in that it does not provide for either

1) the overlap of WriteRequest's with ActualRead's and ActualWrite's, or
2) the overlap of ReadRequest's with ActualWrite's.

It may not be restrictive enough in that it permits the overlapping of ReadRequest's with each other. To satisfy 1) and 2) and, at the same time, make the request's mutually exclusive, synchronization symbols and locks are added as follows:

$$([\text{ ReadRequest }] \; \omega_{R_1} \text{ ActualRead } \sigma_{R_2} \;)^{\circledast}$$
$$\odot$$
$$(\; [\text{ WriteRequest }] \; \omega_{W_1} \text{ ActualWrite } \sigma_{W_2} \;)^{\circledast}$$
$$\odot$$
$$(\; (\; \sigma_{R_1} \; \omega_{R_2} \;)^{\circledast} \cup (\; \sigma_{W_1} \; \omega_{W_2} \;) \;)^{*}$$

3.3.4 Some formal properties of FE's are given in [55]. FE's that do not contain either $\circledast$ or ∞ are equivalent to RE's. It was conjectured in [55] that EE's were more powerful than FE's but that apparently is false since proofs are being circulated that FE's are indeed universal also, e.g. [4].

The analysis and verification possibilities appear to be identical to those of EE's [47,55]. Some of these can be stated as general questions such as:
Does a system exhibit a given behavior?
What are the deadlock and starvation possibilities?
Can a resource request or request sequence be serviced without violating any systems constraints?
In the FE context, these questions become:
Membership problems: Is some string in the language described by an FE, or is it a prefix of some sentence in the language?
Equivalence and inclusion problems: Are the languages described by two FE's identical, or is one language included in the other?
Emptiness problem: Is the language given by an FE empty?
These questions are either undecidable or unacceptably complex to answer (compute) for the general FE or EE case; instead one is forced to deal with restricted subsets or specific applications, much the same way as is done in the PE work.

3.3.5 In [55], small examples illustrating a large number of software description applications were presented. These included modelling of sequential and parallel program control constructs, mappings between programs and FE's, descriptions of multiprogramming systems with centralized and distributed control, and specification of synchronization constraints in reader-writer, bounded and unbounded buffer, and resource priority scheduling systems; in addition, the application to command language definition was first developed. My most recent FE work has been concerned mainly with the new applications of interleaving discussed in Section 3.1.4. FE's are also being used to describe and verify some design and research ideas for single-user, single language distributed systems.

4. Path Expressions and Related Schemes

The term "path expressions" denotes many path notations that are based on the scheme introduced by Campbell and Habermann [13]. PE's were designed to describe execution sequences of operations or procedures, and the notational variations are meant to either simplify or handle some aspects of the description, analysis, and/or implementation of the constraints on operation executions. The PE orientation can be contrasted with that of EE's and FE's which are proposed as more general notational tools. PE's have been embedded directly in data types definitions and have also been used as an abstract "programming" language for systems of processes and resources. After presenting most of the PE notations, I will discuss these two applications in some detail and mention some other PE related work in protection and graphics.

4.1 Path Notations

I first discuss PE's that use RE's only. Then several parallel operators are introduced. Finally, schemes for conditional paths, priorities, and counting are described.

A PE is given by the syntax

<u>path</u> expr <u>end</u>

where expr describes execution sequences that may be indefinitely repeated. The <u>path</u> <u>end</u> is thus interpreted as * bracketting the expr.

4.1.1 <u>Regular PE's</u> (<u>RPE</u>'s) are those that consist only of RE's.

<u>Example 19:</u>

<u>path</u> (Insert Look* Remove) ∪ Look <u>end</u>

This RPE may specify the restrictions of the operations Insert, Remove, and Look of a particular one slot data type. The constraints are that all operations are sequential (mutually exclusive), two Inserts cannot occur without an intervening Remove, each Remove must be preceded by an Insert, and Look's can proceed at any time.

The PE literature employs the symbols ";" and either "+" or "," for concatenation and selection, respectively, but I will stay with the RE symbols of the earlier sections. Restrictions on the RPE's, such as no repeated procedure names, or limited or no use of the interior *, have been proposed to simplify implementation or analysis, e.g. [11,13,32].

4.1.2 When more then one PE is permitted for a description of the same set of operations, the notation becomes more powerful, e.g. [11,21,32]. The interpretation is that operation histories must be compatible with all of the PE's in the multiple path specification. Multiple paths implicitly provide for some parallelism; if a relative ordering between two operations cannot be derived from the description, then the operations may proceed in parallel.

Example 20:

Consider the two RPE's

path Fetch Print end   and path Fetch Store end

The first expression states that Fetch's and Print's must alternate while the second one specifies alternating Fetch's and Store's. This is interpreted to mean that between two successive Fetch's there must be both a Store and a Print, between two successive Store's there must be a Fetch, and between two successive Print's there must be a Fetch. These constraints are roughly equivalent to those in the single SE

(Fetch (Print ⊛ Store))*

The difference is that Print and Store could be executed truly in parallel according to the RPE's. Because the RPE's give no constraints on the relative orderings of Print and Store, any or no ordering is permissible.

4.1.3 Possible parallel instantiations are specified with brackets { }. Thus {expr} describes zero or more parallel instantiations of expr; {expr} is approximately equivalent to the SE expr$^{⊛}$, except that non-interleaved parallelism is also included in the former.

Example 21:

The reader-writer constraints presented in Example 4 are re-expressed as a RPE with { }:

path { Read } ∪ Write end

4.1.4 The { } construct, RE's without *, and a mechanism for describing finite parallel execution are the components of

Open Path Expressions (OPE's)[11,14]; OPE's use a different semantics than the other RE-based schemes. Finite bounded parallelism is specified by the notation:

n : (expr)

where n is an integer and expr is an OPE. The meaning is that up to n simultaneous executions of the procedures specified by expr are possible, provided that any surrounding constraints are also satisfied. The semantics are such that, for example, the simple OPE's

path p end   and   path a ∪ b end

each specify no restrictions whatever on the execution of p, a, or b; they essentially just declare the existence of p, and a and b. However, the OPE

path a b end

states that each b must be preceded by a matching a and contains no constraints on how many a's might be executed; it is then equivalent to {a b} .

Example 22:

1) The OPE

path 1: (Insert Remove) end

is equivalent to the RPE

path Insert Remove end

because it states that only 1 simultaneous execution of the sequences in brackets is possible.

2) The OPE

path 1: ({Read}∪Write) end

gives the same permissible executions as the RPE in Example 21; "1:" distributes through to {Read} and Write.

3) path 3: (1: (Insert) 1: (Remove)) end

This OPE permits up to 3 parallel instantiations of Insert Remove sequences, but Insert's are mutually exclusive and Remove's are mutually exclusive.

OPE's are also of interest because there exists an implementation [12,14] (see next Section). The range of describable behaviors is however not evident.

4.1.5 An explicit parallel operator has been defined for PE's [2,11,21]. A form of interleaving, called a "connected path" has also been proposed [21]. Neither of these operators seem to be used to any extent in the published PE examples but they could be employed for some of the same tasks as the SE ⊙.

4.1.6 Conditional paths [11,21] and predicate paths [2] offer a still finer control over the selection of path elements. As an alternative to the PE selection a ∪ b ∪ ... ∪ c , conditional paths permit a Boolean expression Bi (with no side effects) to be attached to each element i of the selection [19]:

```
    if Ba then a else if Bb then b ... else if Bc then c
    or
    if Ba then a else if Bb then b ... else c    *)
```

In [11], the semantics of the conditional elements is similar to Dijkstra's guarded commands.

Example 23:

Consider a bounded stack of maximum size bound, with operations Push, Pop, and Top, and a data element count that contains the current number of elements in the stack. The operation constraints are given by the PE with conditional elements:

```
    path if count = 0 then Push
    else if count = bound then ( Pop ∪ Top )
    else ( Push ∪ Pop ∪ Top ) end
```

In Andler's proposal for predicate paths [2], the Boolean expression (predicates) may be attached to any path element, and may use only constants and event counters that are associated with path elements; the predicate PE's also follow a guarded command semantics.

*) I am again talking liberties with the syntax of the published notations, as is done throughout the paper.

4.1.7 Priorities may be specified in the selection part of a PE using > and < instead of $\cup$ [11,21].

Example 24:

Suppose the three operations on a memory resource are GetSpace, FreeSpace,and GarbageCollect. Then the PE with priorities

path FreeSpace > GarbageCollect > GetSpace end

reflects the scheduling policy that FreeSpace requests have priority over GarbageCollect's, and GarbageCollect's have priority over GetSpace requests.

4.1.8 A counting facility, called a numeric path element, is used in [16] with the general form $(a - b - \ldots - c)^n$. This has the meaning $(a \cup b \cup \ldots \cup c)$ with the constraint that the relation $\#(a) \geq \#(b) \geq \ldots \geq \#(c) \geq \#(a) - n$ remains invariant, where $\#(x)$ means the number of executions of x.

Example 25:

path (Insert - Remove)^{15} end

This describes the selection (Insert $\cup$ Remove)* with the restrictions that $0 \leq \#(\text{Insert}) - \#(\text{Remove}) \leq 15$, thus modelling, for example, operations on a bounded buffer with 15 slots. An equivalent SE is quite lenthy:

$$\left(\bigcup_{i=0}^{15} (\text{Insert Remove})^{\circledast i} \right)^*$$

where $a^{\circledast 0} = \lambda$, and $a^{\circledast i+1} = (a^{\circledast i} \circledast a)$ for $i \geq 0$.

4.1.9 Several other variations and additions have been suggested. By selecting "options" appropriately, one can easily produce many hundreds of different path notations.

The most "popular" path notations appear to be the following:

1) the original PE's introduced in [13]

These consist of RE's without the internal * and the parallel

{ } without nesting. A procedure name (basic symbol) can only appear once in an expression.

2) RPE's and their restrictions (4.1.1)
3) multiple PE's (4.1.2)
   Each individual PE is an RPE or a restricted RPE.
4) OPE's (4.1.4)

4.1.10 The semantics of some of the notations have been defined more precisely so that implementations, analysis, or verification are possible. The definitional methods include:

1) relatively informal implementation-oriented descriptions.
   It is shown how the procedure constraints contained in a PE can be directly mapped either into Dijkstra P and V operations that are inserted before (prologue) and after (epilogue) the procedure or into more complex prologues and epilogues involving path "state" testing. The former techniques are used in defining the original notation [13] and for OPE's [11,14] while the more complex scheme appears necessary for RPE's [24].
2) Petri net transformations.
   RPE's and their restrictions, and multiple PE's, have all been defined by exhibiting transformations that map the expressions into corresponding Petri nets [11,32,34]. The Petri nets define the PE's in the sense that the set of procedure execution sequences accepted or described by a PE is the set of event sequences generated by the corresponding net.
3) others such as axioms or invariants for program execution sequences constrained by PE's [16], partial ordering of events generated by program computations [2], and formal denotational and axiomatic techniques [7].

The descriptive power of PE's is difficult to establish once one goes beyond RE's. The principal technique has been to relate PE classes to different classes of Petri nets [32,34]. This has also permitted the use of net theory to investigate properties such as deadlock and starvation, e.g. [33,34].

4.2 Object Definitions

4.2.1 The intended application of PE's is in data type specifications. Instead of distributing synchronization primitives throughout the operations of software objects, it is proposed that each object centrally contain PE's that declare the required synchronization constraints on its use. This kind of data object has the form:

```
      type <object name>
           <data declarations>
           path <expr> end
           operations
                procedure <procedure name and definition>
                procedure <procedure name and definition>
                    .
                    .
                    .
                procedure <procedure name and definition>
      end type
```

The <expr> in the PE specifies the restrictions on the executions of the procedures declared in the object.

Example 26:

```
type OneSlotBuffer;
     frame: message ; /* frame is a variable of type message.*/
     path Put  Get end;
     operations
          procedure Put(x: message) ; /* Insert message into buffer. */
          begin frame := x end ;
          procedure Get(x: message) ; /* Remove message from buffer. */
          begin x := frame end ;
end type
```

The PE describes the standard restrictions; that is, Get and Put are critical sections and must follow in sequence. It is not necessary to either surround calls of Get and Put by synchronization operations or to distribute those operations through Get and Put.

4.2.2 The idea of removing synchronization specifications from operations and user programs, and inserting them as PE's in object declarations is extremely attractive. This approach is contrasted with the use of monitors where each procedure is implicitly a critical section and synchronization primitives (wait and signal) are distributed through the procedures. However, while PE's embedded in data types provide elegant solutions to many synchronization problems, it is not evident that they are powerful enough to handle <u>all</u> such problems; it appears that one may eventually be forced to extend PE's to a universal programming or description language or provide some "escape" to express synchronization outside the PE mechanism. These remarks apply primarily to systems where the PE is part of the <u>programming</u> language as opposed to the <u>specification</u> language.

The literature contains many examples of object (type) definitions with PE's used as above, including ring buffers (bounded buffers), Dijkstra P and V, stack, alarm clock service, disk scheduler, message passing, and various other resource allocators, e.g. [3,11,12,13,21]. PE's in abstract data types also appear to simplify verification, primarily because (when) the invariants derived from the PE may be almost identical to the invariant to be proven [1,16].

4.2.3 There exist two programming implementations of data types with PE's. OPE's have been included in the Path Pascal system recently described in [12,14]. This system extends Pascal with processes, OPE's, and encapsulated data objects. There is also an implementation reported in [2] which introduces types into Algol 68 [3]. RPE's plus predicates (Section 4.1.6), called predicate path expressions (PPE's), are used. The implementations are important to test the practicality of the PE ideas for a <u>systems programming</u> language; they are also of great potential value in design <u>simulation</u> and for determining the benefits and limitations of PE's in larger systems contexts.

4.3 <u>Some PE Related Work</u>

4.3.1 Some other recent related proposals are also of interest. In the PE work, the operations of the data type may be executed by _any_ process provided the PE is not violated. It is often the case, however, that some selectivity is desirable to ensure that users employ the type properly.

Example 27:

Suppose a data object has operations startread and endread constrained to execute in sequence by the PE

path startread endread _end_

A startread might be executed by a user U_1 followed by the execution of endread by a different user U_2. While this sequence may be acceptable in some applications, there are many where it would be incorrect; only the user that executes startread should be able to subsequently execute endread.

RE's called _access right expressions_ (_ARE_'s) are suggested in [27] to handle the above kind of situation. A set of ARE's is given in the definition of an object. When a user program U declares an _instance_ of the type, it also specifies a subset of the object's ARE's. U can then only execute operations according to the sequencing declared in its instance. Thus, in the example above, _each_ user could get an instance of the ARE (startread endread)*.

4.3.2 A variation of PE's has also been suggested in a completely different domain - the specification of logical graphics input devices [8,9,10]. The sequences of input events comprising a logical device are described by a PE-like notation, an _input expression_ (_IE_), that appears as part of the device definition. The device definition, a data object, contains code that is invoked whenever an input sequence satisfies the expression in the IE. These ideas have recently been implemented as an extension of the C language under the UNIX system [37].

Example 28:

input (Key("+") Key(CR)) $\cup$ Button(6) $\cup$ Pick("menu",5) _end_

The above IE describes three simple input event sequence alternatives:

1) a '+' followed by a carriage return (CR) entered through a keyboard,

2) the single event corresponding to depressing button number 6, and
3) the single event corresponding to pointing (on a display screen) at item number 5 of the segment named "menu".

The IE is satisfied if any of these sequences occur; the code associated with the IE would then be executed.

The IE's are regular expressions plus priorities, a parallel operator, and conditional elements. The basic symbols denote either input primitives with parameters, or higher level logical devices defined previously. This approach has some similarities with my work on command languages mentioned in Section 3.1.4.

4.4 System Descriptions: Path and Process Expressions

4.4.1 Another major class of applications employs PE ideas to describe both the flow through the procedures of each process in a (sub)system *and* the constraints on the procedure executions [11, 32,33,34,60]. The latter PE's are essentially abstractions of data type definitions, i.e. resource objects, and may consist of multiple PE's; the process paths approximate the control flow of processes.

Example 29:

```
begin path lock unlock end
  p1: process compute1 lock CS1 unlock end
  p2: process compute2 lock CS2 unlock end
end
```

The path specifies a locking mechanism (binary semaphores), restricting the lock and unlock calls to the sequences described by (lock unlock)*. p_1 and p_2 are cyclic processes with cyclic execution sequences given, for example, by the FE's

$$(\text{compute1 lock CS1 unlock})^\infty \text{ and } (\text{compute2 lock CS2 unlock })^\infty$$

The process flows are constrained by the path declarations so that CS1 and CS2 are mutually exclusive (critical sections); compute1 may proceed in parallel with any of the operations of p_2 and analogously for compute2.

A program in the basic path/process notation is then a collection of paths followed by a collection of processes. The multiple paths are interpreted as in Section 4.1.2, and the processes are independent entities with execution sequences subject to the constraints of the paths. RE's and restricted RE's are used for the path and process notations. The semantics of the path/process languages are described by Petri nets and are related to various classes of nets. One aim of this work is to use net theory as a basis for proving properties of the languages and programs written in the languages. Much emphasis has been placed on the notion of "adequacy"; this is a type of correctness, defined in net theory, that implies freedom from deadlock.

4.4.2 The basic path/process notation is extended to permit the definition of classes of paths and their instantiation[11,32,34,60]. The most recent development is the COSY language which contains a macro facility with elaborate path/process generators[34,60].

Example 30: (from [34])

Below is a COSY program:

```
begin array DEPOSIT,REMOVE(n)
        [path DEPOSIT(i) REMOVE(i) end [i]/1,n,1]
        [ process ∪(DEPOSIT) end [i]/1,m,1]
        [ process ∪(REMOVE) end [i]/1,k,1]
end
```

The path declarations "macro" expands to n paths:

```
path DEPOSIT(1) REMOVE(1) end ... path DEPOSIT(n) REMOVE(n) end
```

The array declares each DEPOSIT(i) and REMOVE(i), i=1,...,n as a separate distinct operation.

The notation ∪(DEPOSIT) is an abbreviation for

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

The first process then expands to m processes, each a copy of

```
process DEPOSIT(1) ∪ ... ∪ DEPOSIT(n) end
```

Similar expansions hold for the second process. The COSY program then defines m+k processes, subject to restrictions given by n paths. It models an n slot buffer that permits a process to DEPOSIT into an empty slot and REMOVE from a full one (one that had been previously DEPOSIT'ed).

The COSY notation has been used to describe many small systems including buffering programs with different disciplines, extended (and bounded) semaphores, and a simple spooling system [33,34,58,59]. Some suggestions are given in [58] for extending the notation to handle unbounded elements, for example, unbounded counters. As in the other PE notations, COSY does not have the full power of a general programming language (Turing machine).

5. Assessment and Future Directions

5.1 Many regular expression based schemes are proposed as design aids for describing sequential and concurrent software. The numerous specification examples appearing in published reports demonstrate, at least on a small scale, that convenient and tractable descriptions can be produced with these notations; in particular, standard, and difficult, synchronization problems, such as those, involving critical sections, reader-writers, producer-consumers, buffering, and priority scheduling, have surprisingly simple descriptions. In addition, the notations have resulted in some new approaches in systems analysis, and programming.

I believe that the principal contributions, and novel and useful ideas of these works are the following:

1) The behavior of a system should (can?) be modelled in terms of sequences of events, flows, or operations that may occur during execution. Much of systems design is concerned, directly or indirectly, with the specification of permissible sequences.
2) Given 1), RE's are a natural notation on which to base a non-procedural design language.
3) Shuffling operators can express concurrent and interleaved behaviors, and are a natural extension to RE's.
4) Given 1), path expressions are an attractive way to describe synchronization constraints on operations in resource objects.
5) The resulting EE, FE, and PE notations have theoretical properties and underlying models of computing that are interesting in their own right.

5.2 The EE and FE notations are similar in that they both use the same shuffling operators (SE's); however, they have different synchronization schemes, and a distinction is made between indefinite repetition (*) and cyclic or infinite repetition (∞) in FE's. Examples can be found where either scheme provides a simpler specification than the other. It is difficult to compare the analysis possibilities of the two notations, but software analysis applications have been more extensively investigated in the EE case.

PE's, as a general class of notations, have several properties that distinguish them from EE's and FE's. The extension of RE's that appear in PE's are completely different than the synchronization mechanisms of EE's and FE's. The parallel operator { } can be modelled by $\circledast$ but interleaving is not the same as potentially asynchronous concurrency; multiple paths also have parallelism by default. In both cases, the total orderings of elements in FE and EE descriptions are replaced by partial orderings. None of the PE schemes are universal while EE's and FE's are capable of describing <u>any</u> computable constraint. The specific interpretation of PE elements as procedure executions and the programming language flavour of some of the extensions, e.g. conditional elements, also permit some descriptions that are not easy to formulate in EE's or FE's. While FE's and EE's are not meant to be implemented in programming languages, PE's are used in both the design and programming stages of software development.

5.3 Research with all of these schemes has lead to some notational issues that should be further analyzed:

1) bounded vs unbounded parallelism.
   Should unbounded parallelism be expressible in a software design notation? Is an operator such as $\circledast$ or { } necessary? Bounded schemes are more tractable theoretically (e.g. Petri net models) but unbounded methods appear to be more natural when exact numbers of resources or processes are not known, or when they may vary dynamically.

2) interleave vs concurrency

Interleaving is an adequate model for concurrency in most applications but it can well be argued, at least philosophically, that it does not always capture the notion of concurrency, especially for distributed asynchronous systems. Parallel operators such as the PE { } are one solution but they appear to be difficult to formally manipulate. Notations based directly on partial orderings of events offer another possible approach that might be developed further into a description notation [19,29,38]. Here for two events a and b, either a precedes b ($a \rightarrow b$), b precedes a ($b \rightarrow a$), or neither ($a \not\rightarrow b$ and $b \not\rightarrow a$); in the last case, the events are concurrent.

3) limited power.

How descriptively powerful should a design notation be? One apparent reason for the each proposed extension of PE's is the discovery of a new problem that requires the extension; this is inevitable unless the notation is universal. A design notation does not have to be universal necessarily, especially if it describes most practical behaviors conveniently and precisely, and provides reasonable approximations to others.

5.4 Design notations are only useful if they facilitate the analysis and implementation phases of software development. I have referred to some of the work in these other phases but it is fair to say that the new notations have not yet been used as practical tools. On the other hand, they do provide a precise software specification language that can be clearly independent of implementations as compared with procedural design languages.

The many examples and theoretical results show that the RE based notations are promising. Some practical experiences in software design and development would seem to be the next logical step. The experiences obtained with the several implementations should be of value in further assessing the notations [3,12,37,50].

5.5 There are also some new description applications worth pursuing. The problems of distributed systems and their message communications are still being formulated, and the RE description notations may be useful for problem and design specifications. Coroutines have been recognized as a fundamental control structure for both simulating concurrent activities and for naturally interleaved tasks; FE's and EE's, which can describe coroutine-like interleaving, should be studied as a design notation for coroutine-based software. The uses of interleaving in command language specifications [55,56] and PE's in input processing [8,9,10] are still relatively undeveloped. Finally, there are some protection applications, such as the ARE proposal [27], where constraints on sequences of operations, messages, and other entities are required; two of these are access control lists for files - who can use a particular file and how, and limitations on process operations such as those expressed by capabilities.

References and Bibliography

(The bibliographic items are tagged EE, FE, and/or PE, depending on whether they are concerned with event expressions, flow expressions, or path expressions, respectively).

1. S. Andler. Synchronization primitives and the verification of concurrent programs. Proc. 2nd Internat. Symp. on Operating Systems, IRIA, Le Chesnay, France (October 1978). {PE}.

2. S. Andler. Predicate path expressions. Proc. 6th Annual ACM Symp. on Principles of Programming Languages, San Antonio, Texas (January 1979), pp 226-236. {PE}.

3. S.Andler and P.G. Hibbard. Types in Algol 68. Proc. 5th Annual III Conf. on the Implementation and Design of Algorithmic Languages, IRIA, Le Chesnay, France (May 1977), pp 124-144. {PE}.

4. T.Araki and N. Tokura. Flow languages are recursively enumerable. Programming Languages Group Memo No. 78-01. Dept. of Information and Computer Sciences, Osaka University, Japan (November 1978). {FE}.

5. T. Araki and N. Tokura. Undecidability of the equivalence problem for flow expressions. (1978). Accepted for publication in Trans IECE Japan (in Japanese). {FE}.

6. T.Araki, T. Kagimosa, and N. Tokura. Relations of flow languages to Petri net languages. Programming Languages Group Memo No. 79-01, Dept. of Information and Computer Sciences, Osaka University, Japan (February 1979). {FE}.

7. V. Berzins. Denotational and axiomatic definitions for path expressions. Computation Structures Group Memo 153-1, Lab. for Computer Science, MIT, Cambridge, Mass. (November 1977). {PE}.

8. J. van den Bos. Definition and use of higher level graphics input tools. Proc. SIGGRAPH'78, Computer Graphics 12, 3 (August 1978), pp 38-42 {PE}.

9. J. van den Bos. High-level graphics input tools and their semantics. Paper for the IFIP W.G. 5.2 Workshop on Methodology of Interaction, Seillac, France, May 1979 (to be published by North-Holland).{PE}.

10. J. van den Bos. Input tools - a new language construct for input-driven programs. Report No. 12, Informatica / Computer Graphics, Nijmegen University, Nijmegen, The Netherlands (March 1979). (to be presented at the Europ. IFIP Conf., London, October 1979). {PE}.

11. R.H. Campbell. Path expressions: a technique for specifying process synchronization. Ph.D. Thesis, Computing Lab., the University of Newcastle Upon Tyne, England (August 1976). (Also reprinted as UIUCDCS-R-77-863, Dept.of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL (May 1977).) {PE}.

12. R.H. Campbell, I.B. Greenberg, and T.J. Müller. Path Pascal User Manual. TR # R-79-960, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL (Feb. 1979). {PE}.

13. R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. Lecture Notes in Computer Science 16, Springer Verlag, Heidelberg (1974), pp 89-102. {PE}.

14. R.H. Campbell and R.B. Kolstad. Path expressions in Pascal. Dept. of Computer Science, University of Illinois at Champaign-Urbana, Urbana, IL (January 1979). {PE}.

15. D. Comte, G. Durrieu, O. Gelly, A. Plas, and J.C. Syre. Parallelism, control and synchronization expression in a single assignment language. SIGPLAN Notices 13, 1 (January 1978), pp 25-33. {PE}.

16. L. Flon and A.N. Habermann. Towards the construction of verifiable software systems. Proc. ACM Conf. on Data, SIGPLAN Notices 8, 2 (March 1976), pp 141-148. {PE}.

17. S. Ginsburg. The Mathematical Theory of Context-Free Languages McGraw-Hill, NY (1966), p. 108.

18. S. Ginsburg and E.H. Spanier. Mappings of languages by two-tape devices. JACM 12, (1965), pp 423-434.

19. I. Greif. A language for formal problem specification. Comm. ACM 20, 12 (December 1977), pp 931-935.

20. A.N. Habermann. Operations on shared data controlled by function modules in type definitions. Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA (September 1973). {PE}.

21. A.N. Habermann. Path expressions. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (June 1975). {PE}.

22. A.N. Habermann. On the timing restrictions of concurrent processes. Proc. 4th Texas Conf. on Computer Systems, Austin, Texas (November 1975). {PE}.

23. A.N. Habermann. Introduction To Operating System Design. SRA, Chicago (1976). {PE}.

24. A.N. Habermann. Implementation of regular path expressions. Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA (1979). {PE}.

25. D. Ingalls. The Smalltalk-76 programming language. Proc.Fifth Annual ACM Symp. on Principles of Programming Languages, (January 1978).

26. T.Kagimasa, T. Araki, and N. Tokura. The descriptive power of flow expressions. Papers of Technical Group on Automata and Languages, AL 78-77, IECE Japan (January 1979) (in Japanese). {FE}.

27. R.B. Kieburtz and A. Silberschatz. Access-right expressions. TR # 44, U.T.D. Programs in Math. Science, University of Texas at Dallas, Dallas, Texas (1978).

28. T. Kimura. Behavioral abstraction of communicating sequential processes, Dept. of Computer Science, Washington Univ., St. Louis, Missouri (January 1979). {EE,FE}.

29. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Comm. ACM 21, 7(July 1978), pp 558-564.

30. P.E. Lauer, E. Best, and M.W. Shields. On the problem of achieving adequacy of concurrent programs. Proc. IFIP Working Conf. on Formal Description of Programming Concepts, North Holland (August 1977). {PE}.

31. P.E. Lauer and R.H. Campbell. A description of path expressions by Petri nets. Proc. 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, CA (Jan. 1975), pp 95-105. {PE}.

32. P.E. Lauer and R.H. Campbell. Formal semantics of a class of high level primitives for coordinating concurrent processes. Acta Informatica 5, (1975), 297-332. {PE}.

33. P.E. Lauer and M.W. Shields. Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts. SIGPLAN Notices 13, 12 (1978), pp 41-59. {PE}.

34. P.E. Lauer, M.W. Shields, and E. Best. On the design and certification of asynchronous systems of processes; ASM/45 Part 2: Formal theory of the basic COSY notation (March 1978); ASM/49 Part 1: COSY - a system specification language based on paths and processes (June 1978). Computing Lab., The University of Newcastle Upon Tyne, England. {PE}.

35. M.L. Minsky. Computation: Finite and Infinite Machines. Prentice-Hall, NJ (1967).

36. W.F. Ogden, W.E. Riddle, and W.C. Rounds. Complexity of expressions allowing concurrency. Proc. Fifth Annual ACM Symp. on Principles of Programming Languages (January 1978). {EE, PE}.

37. R. Plasmeijer, J. van den.Bos, and J. Stroet. An implementation of high-level graphics tools. Informatica / Computer Graphics, Univ. of Nijmegen, Nijmegen, The Netherlands (Dec. 1978). {PE}.

38. D.P. Reed and R.K. Kanodia. Synchronization with eventcounts and sequencers. Comm. ACM 22, 2 (February 1979), p 115-123.

39. W.E. Riddle. Modelling and analysis of supervisory systems. Ph.D. Thesis, Computer Science Dept., Stanford University, Stanford, CA (March 1972). {EE}.

40. W.E. Riddle. The hierarchical modelling of operating system structure and behavior. Proc. ACM National Conf. (August 1972). {EE}.

41. W.E. Riddle. A method for the description and analysis of complex software systems. Proc. ACM SIGPLAN-SIGOPS Conf. on Programming Languages and Operating Systems, SIGPLAN Notices 8, 9 (Sept. 1973), pp 133-136. {EE}.

42. W.E. Riddle. A design methodology for complex software systems. Proc. Second Texas Conf. on Computing, Austin, Texas (Nov. 1973) pp 22/1-22/8.

43. W.E. Riddle. The equivalence of Petri nets and message transmission models. SRM/97, Computing Lab., The University of Newcastle Upon Tyne, England (August 1974). {EE}.

44. W.E. Riddle. Message transfer expressions and their use in specifying synchronization constraints. RSSM/1, Dept. of Computer and Comm. Sciences, University of Michigan, Ann Arbor, MI (September 1974). {EE}.

45. W.E. Riddle. The translation of path expressions into message transfer expressions. RSSM/2, Dept. of Computer and Comm. Sciences, University of Michigan, Ann Arbor, MI (September 1974). {EE, PE}.

46. W.E. Riddle. Computer augmented design of complex software systems. Proc. Fourth Texas Conf. on Computing Systems, Austin, Texas (November 1975), pp 3A/2.1-3A/2.12. {EE}.

47. W.E. Riddle. Software systems modelling and analysis. RSSM/25, Dept. of Computer and Comm. Sciences, University of Michigan, Ann Arbor, MI (July 1976). (To appear in the Journal of Computer Languages). {EE}.

48. W.E. Riddle, J.H. Sayler, A.R. Segal, A.M. Stavely, and J.C. Wileden. A description scheme to aid the design of collections of concurrent processes. Proc. 1978 National Computer Conf., (1978), pp. 549-554. {EE}.

49. W.E. Riddle, J.H. Sayler, A.R. Segal, A.M. Stavely, and J.C. Wileden. Abstract monitor types. Proc. Specification of Reliable Software Conf., Boston, (April 1979).{EE}.

50. W.E. Riddle, J.H. Sayler, A.R. Segal, and J. Wileden. An introduction to the DREAM software design system. Software Engineering Notes 2, 4 (July 1977), pp 11-24.{EE}.

51. W.E. Riddle, J.C. Wileden, J.H. Sayler, A.R. Segal, and A.M. Stavely. Behavior modelling during software design. IEEE Trans. on Software Engineering SE-4, 4 (July 1978), 283-292. {EE}.

52. J. Sanguinetti. Performance prediction in an operating system design methodology. Ph.D. Thesis. Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI (May 1977). {EE}.

53. J. Sanguinetti. A formal technique for analyzing the performance of complex systems. Proc. Computer Performance Evaluation Users Group Conf., Boston, Mass. (October 1978). {EE}.

54. A.C. Shaw. Systems design and documentation using path descriptions. Proc. 1975 Sagamore Computer Conf. on Parallel Processing, IEEE Computer Society, Silver Spring, MD (1975), pp 180-181. {FE}.

55. A.C. Shaw. Software descriptions with flow expressions. IEEE Trans. on Software Engineering SE-4, 3 (May 1978), 242-254. {FE}.

56. A.C. Shaw. On the specification of graphics command languages and their processors. TR 29, Institut fuer Informatik, ETH, Zurich (January 1979) (Paper for the IFIP W.G. 5.2 Workshop on Methodology of Interaction, Seillac, France, May 1979) (to be published by North-Holland.) {FE}.

57. M.W. Shields and P.E. Lauer. On the abstract specification and formal analysis of synchronization properties of concurrent systems. Proc. International Conf. on Math. Studies of Information Processing, Springer-Verlag (1978). {PE}.

58. M.W. Shields and P.E. Lauer. A formal semantics for concurrent systems. To appear in Proc. 6th International Colloq. Automata, Languages, and Programming (July 1979), Graz, Austria, Springer-Verlag. {PE}.

59. P.R. Torrigiani. Synchronic aspects of data types: construction of a non-algorithmic solution of the banker's problem. Proc. ECI 78, Lecture Notes in Computer Science 65, Springer-Verlag (1978), pp. 560-583. {PE}.

60. P.R. Torrigiani and P.E. Lauer. An object oriented notation for path expressions. AICA 1977, Vol. 3, Software Methodologies (Oct. 1977), pp. 349-371. {PE}.

61. M. Welter. Counter expressions. RSSM/24, Dept. of Computer and Comm. Sciences, University of Michigan, Ann Arbor, MI (August 1976). {EE}.

62. J.C. Wileden. Modelling parallel systems with dynamic structures. Ph.D. Thesis, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, MI (1978). (also published as TR 78-4, Computer and Infor. Science Dept., Univ. of Massachusetts, Amherst, Mass. (January 1978).) {EE}.

63. M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon. Principles of Software Engineering and Design. Prentice-Hall, NJ (1979), Chapter 4. {FE}.

Index to Symbols and Abbreviations

(The Section entry for each item points to the section in the paper where the item is defined.)

Symbol or Abbreviation	Meaning	Section
ARE	access right expression	4.3.1
EE	event expression	3.2
FE	flow expression	3.3
IE	input expression	4.3.2
MTE	message transfer expression	3.2
OPE	open path expression	4.1.4
PE	path expression	4
RE	regular expression	2
RPE	regular path expression	4.1.1
SE	shuffle expression	3.1
$\cup$	RE selection	2.1
$*$	Kleene star	2.1
λ	empty string	2.1
$\odot$	SE shuffle operator	3.1.1
$\circledast$	closure of $\odot$	3.1.1
$@, \overline{@}$	EE synchronization symbols	3.2.1
∞	FE cyclic operator	3.3.1
[,]	FE lock	3.3.2
σ, ω	FE signal and wait	3.3.2
{ }	PE simultaneous execution	4.1.3

Berichte des Instituts für Informatik

*Nr. 1	N.Wirth:	The Programming Language PASCAL
*Nr. 2	N.Wirth:	Program development by step-wise refinement
Nr. 3	P.Läuchli:	Reduktion elektrischer Netzwerke und Gauss'sche Elimination
Nr. 4	W.Gander, A.Mazzario:	Numerische Prozeduren I
*Nr. 5	N.Wirth:	The Programming Language PASCAL (Revised Report)
*Nr. 6	C.A.R.Hoare, N.Wirth:	An Axiomatic Definition of the Language PASCAL
Nr. 7	W.Gander, A.Mazzario:	Numerische Prozeduren II
Nr. 8	E.Engeler, E.Wiedmer, E.Zachos:	Ein Einblick in die Theorie der Berechnungen
*Nr. 9	H.P.Frei:	Computer Aided Instruction: The Author Language and the System THALES
*Nr.10	K.V.Nori, U.Ammann, K.Jensen, H.H.Nägeli, Ch.Jacobi:	The PASCAL 'P' Compiler: Implementation Notes (Revised Edition)
Nr.11	G.I.Ugron, F.R.Lüthi:	Das Informations-System ELSBETH
*Nr.12	N.Wirth:	PASCAL-S: A subset and its Implementation
*Nr.13	U.Ammann:	Code Generation in a PASCAL Compiler
Nr.14	K.Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
*Nr.15	E.Engeler:	Structural Relations between Programs and Problems
Nr.16	W.Bucher:	A contribution to solving large linear problems
Nr.17	N.Wirth:	Programming languages: what to demand and how to access them and Professor Cleverbyte's visit to heaven
*Nr.18	N.Wirth:	MODULA: A language for modular multiprogramming
*Nr.19	N.Wirth:	The use of MODULA and Design and Implementation of MODULA
Nr.20	E.Wiedmer:	Exaktes Rechnen mit reellen Zahlen