Modeling of Distributed Systems by Concurrent Regular Expressions

Vijay K. Garg
Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712

We propose an algebraic model called concurrent regular expressions for modeling and analysis of distributed systems. These expressions extend regular expressions with four operators - interleaving, interleaving closure, synchronous composition and renaming. Their expressive power is equivalent to those of Petri nets, and therefore they are more general than Path expressions and COSY expressions.

1 Introduction

As concurrent systems are difficult to design, the simplest of them can have subtle errors. To avoid these errors, we need to capture essential aspects of a system in a model and then analyze it for correctness. Models for concurrent systems that can be analyzed automatically have less expressive power than programming languages. They can be categorized roughly into two groups: *algebra based* and *transition based* models. The algebra based models consist of operators and a set of primitive behaviors. Expressions are built hierarchically by applying operators on sub-expressions. Examples of such models are path expressions [Lauer 75], behavior expressions [Milner 80] and extended regular expressions. Examples of specification languages and systems based on such models are Path Pascal [Campbell 79], CCS [Milner 80] and Paisley [Zave 85]. Some of the commonly asked questions in such formal systems are: "Is s a possible trace of the concurrent system under analysis?", and "Is S_1, a concurrent system, the same as another concurrent system S_2?"

In the transition based models the behavior of a system is generally modeled as a sequence of configurations of an automaton. Examples of the *transition based* models are finite state machines [Hopcroft 79], S/R Model [Aggarwal 87], UCLA graphs [Cerf 72], and Petri nets [Reisig 85]. Examples of modeling and analysis tools based on these models are Spanner [Aggarwal 87], Affirm [Gerhart 80] and PROTEAN [Billington 88].

Algebraic systems promote hierarchical description and verification, whereas transition based models have the advantage that they are graphical in nature. For this reason, it is sometimes easier to use an algebraic description, and othertimes a transition-based description. *We believe that a formal description technique should support both styles of descriptions.* In this paper, we propose an algebraic model called concurrent regular expressions for modeling of concurrent systems. These expressions can be converted automatically to Petri nets, and thus all analysis techniques that are applicable to Petri nets can be used. Conversely, any Petri net can be converted to a concurrent regular expression providing further insights into its language.

All the existing models can also be classified according to their inherent expressive power. For example a finite state machine is inherently less expressive than a Petri net. However, the gain in expressive power comes at the expense of analyzability. A complex system may consist of many components requiring varying expressive power. *We believe*

that a formal description technique should support models of different expressive powers under a common framework. An example of such a description technique for syntax specification is Chomsky hierarchy of models based on grammar. A similar hierarchy is required for formal description of distributed systems. The model of concurrent regular expressions provides such a hierarchy. A regular expression is less expressive than a unit expression which, in turn, is less expressive than a concurrent regular expression.

As mentioned earlier, there are many existing algebraic models for specification of concurrent systems. CCS[Milner 80], CSP[Hoare 85] and FRP[Inan 88] These models do not have any equivalent transition based model. Similarly, they do not support a hierarchy of models like we do. Path expressions[Lauer 79] were shown to be translatable to Petri nets, and thus analyzable for reachability properties [Kosaraju 82, Mayr 84,Karp 69]. Concurrent regular expressions are more general than Path expressions as they are equivalent to Petri nets [Garg 88].

We have used interleaving semantics rather than true concurrency as advocated by [Pratt 81] and [Reisig 85]. This assumption is in agreement with CSP[Hoare 85] and CCS[Milner 81]. In this paper, we have further restricted ourselves to modeling deterministic systems so that the languages are sufficient for defining behaviors of a concurrent system. We have purposely restricted ourselves from defining finer semantics, such as failures[Hoare 85], and synchronization trees[Milner 81], as the purpose of this paper is to introduce a basic model to which these concepts can be added later. In particular, it is easy to add a non-deterministic *or* operator and failure semantics[Hoare 85].

Concurrent regular expressions (cre) use operators that arise naturally in modeling concurrent systems such as interleaving and synchronous composition. The expressive power of cre's is increased beyond that of regular expressions by means of an operator called interleaving closure denoted by α. Based on this operator we introduce the notion of interleaving-closed (i-closed) sets and study their properties. The concept of i-closed set is useful because it guarantees that if every customer engages in a legal sequence of action than a sequence of actions with any number of customers will also be legal no matter what the interleaving. This is a desirable property of a system as it tells us that the system can logically handle any number of customers and treat them independently. We provide a method to construct interleaving-closed sets.

This paper is organized as follows. Section 2 defines concurrent regular expressions. It also describes the properties of operators used in the definition. Section 3 gives some examples of use of cre's for modeling distributed systems and compares the modeling convenience of concurrent regular expressions with Petri nets. Section 4 presents the properties of interleaving-closed sets. Section 5 compares the class of languages defined by concurrent regular expressions with regular, context-free and Petri net recognizable languages.

2 Concurrent Regular Expressions

We use languages as the means for defining behaviors of a concurrent system. A language is defined over an alphabet and therefore two languages consisting of the same strings but defined over different alphabet sets will be considered different. For example, null languages defined over Σ_1 and Σ_2 are considered different. We will

generally indicate the set over which the language is defined, but may omit it if clear from the context.

We next define operators required for definition of concurrent regular expressions.

2.1 Choice, Concatenation, Kleene Closure

These are the usual regular expression operators. *Choice* denoted by "+" is defined as follows. Let L_1 and L_2 be two languages defined over Σ_1 and Σ_2 then
$A + B = A \cup B$ defined over $\Sigma_1 \cup \Sigma_2$.
This operator is useful for modeling the choice that a process or an agent may make.
The *Concatenation* of two languages (denoted by .) is defined based on usual concatenation of two strings as
$L_1.L_2 = \{x_1x_2 | x_1 \in L_1, x_2 \in L_2\}$
This operator is useful to capture the notion of a sequence of action followed by another sequence. The *Kleene closure* of a set A is defined as
$A^* = \bigcup_{i=0,1,..} A^i$
where $A^i = A.A...i\ times$
This operator is useful for modeling the situations in which some sequence can be repeated any number of times. For details of these operators, the reader is referred to [Hopcroft 79].

2.2 Interleaving

To define concurrent operations, it is especially useful to be able to specify the interleaving of two sequences. Consider for example the behavior of two independent vending machines VM1 and VM2. The behavior of VM1 may be defined as $(coin.choc)*$ and the behavior of VM2 as $(coin.coffee)*$. Then the behavior of the entire system would be an interleaving of VM1 and VM2. With this motivation, we define an operator called interleaving, denoted by ||. Interleaving is formally defined as follows:
$a||\epsilon = \epsilon||a = a \qquad \forall a \in \Sigma$
$a.s||b.t = a.(s||b.t) \cup b.(a.s||t) \quad \forall a, b \in \Sigma, s, t \in \Sigma*$
Thus, $ab||ac = \{abac, aabc, aacb, acab\}$.
This definition can be extended to interleaving between two sets in a natural way, i.e.
A || B = $\{w | \exists s \in A \wedge t \in B, w \in s||t\}$

For example, consider two sets A and B as follows: A = $\{ab\}$ and B = $\{ba\}$ then A || B = $\{abba, abab, baab, baba\}$.

Note that similar to A || B, we also get a set A || A = $\{aabb, abab\}$. We denote A || A by $A^{(2)}$. We use parentheses in the exponent to distinguish it from the traditional use of the exponent i.e. $A^2 = A.A$.

Interleaving satisfies the following properties:
(1) Interleaving is commutative, i.e., A || B = B || A
(2) Interleaving is associative, i.e., A || (B || C) = (A || B) || C
(3) Epsilon is the identity of interleaving, i.e., A ||{ϵ} = A
(4) The null set is the zero of interleaving, i.e., A ||$\phi = \phi$
(5) Interleaving distributes over choice, i.e., (A+B) || C = (A || C)+(B || C)

This operator, however, does not increase the modeling power of concurrent regular expressions as shown by the following Lemma.

Lemma 1: Any expression that uses || can be reduced to a regular expression without || .

Proof: This follows from the equivalence between finite state machines and regular expressions and the fact that the interleaving of two finite state machines can also be simulated by a finite state machine[Hopcroft 79]. Δ

2.3 Alpha-closure

Consider the behavior of people arriving at a supermarket. We assume that the population of people is infinite. If each person CUST is defined as $(enter.buy.leave)$, then the behavior of the entire population is defined as interleaving of any number of people. With this motivation, we define an analogue of a Kleene-Closure for the interleaving operator, α-closure of a set A, as follows: $A^{\alpha} = \bigcup_{i=0,1,..} A^{(i)}$.
Then if # (a,w) mean the number of occurrences of symbol a in the string w, the interpretation of $CUST^{\alpha}$ is as follows:
$CUST^{\alpha} = \{w | \forall\, prefixes\, s\, of\, w, \#(enter, s) \geq \#(buy, s) \geq \#(leave, s)$, and $\#(enter, w) = \#(buy, w) = \#(leave, w)\}$

Note the difference between Kleene closure and alpha closure. The language shown above cannot be accepted by a finite state machine. This can be shown by the use of the pumping lemma for finite state machines [Hopcroft 79]. We conclude that alpha closure can not be expressed using ordinary regular expression operators.

Intuitively, the alpha closure lets us model the behavior of an unbounded number of identical independent sequential agents. Alpha-closure satisfies the following properties:
1) $A^{\alpha\alpha} = A^{\alpha}$ (idempotence)
2) $(A*)^{\alpha} = A^{\alpha}$ (absorption of *)
3) $(A + B)^{\alpha} = A^{\alpha} || B^{\alpha}$

2.4 Synchronous Composition

To provide synchronization between multiple systems, we define a composition operator denoted by []. Intuitively, this operator ensures that all events that belong to two sets occur simultaneously. For example consider a vending machine VM described by the expression $(coin.choc)*$. If a customer CUST wants a piece of chocolate he must insert a coin. Thus the event *coin* is shared between VM and CUST. The complete system is represented by VM[]CUST which requires that any shared event must belong to both VM and CUST. Formally,

A[]B =$\{w | w/\Sigma_A \in A, w/\Sigma_B \in B\}$

where w/S denotes the restriction of the string w to the symbols in S. For example, $acab/\{a, b\} = aab$ and $acab/\{b, c\} = cb$. If A = $\{ab\}$ and B = $\{ba\}$, then A[]B = ϕ as there cannot be any string that satisfies ordering imposed by both A and B. Consider another set C = $\{ac\}$. Then A[]C = $\{abc, acb\}$.

Many properties of [] are the same as those of the intersection of two sets. Indeed, if both operands have the same alphabet then [] is identical to intersection.

(1) A[]A = A *(Idempotence)*

(2) A[]B = B[]A *(Commutativity)*
(3) A[](B[]C) = (A[]B)[]C *(Associativity)*
(4) A[]NULL = NULL, NULL = (Σ_A, ϕ) *(zero of* [])
(5) A[]MAX = A, MAX = $(\Sigma_A, \Sigma_A{}^*)$ *(identity of* [])
(6) A[](B+C) = (A[]B)+(A[]C) *(Distributivity over +)*

2.5 Renaming

In many applications, it is useful to be able to rename the event symbols of a process. The renaming can be useful in the following situations:

- Hiding: We may want some events to be internal to a process. We can do so by means of renaming these event symbols to ϵ.
- Partial Observation: We may want to model the situation in which two symbols a and b look identical to the environment. In such cases we may rename both of these symbols with a common name such as c.
- Similar processes: Many system often have "similar" processes. Instead of defining each one of them individually, we may define a generic process which is then transformed to the required process by renaming operator.

Let L_1 be a language defined over Σ_1. Let σ represent a function from Σ_1 to $\Sigma_2 \cup \{\epsilon\}$. Then $\sigma(L_1)$ is a language defined over $\sigma(\Sigma_1)$ defined as follows:
$\sigma(L_1) = \{\sigma(x) | x \in L_1\}$. A renaming operator labels every symbol a in the string by $\sigma(a)$. We leave it to readers to derive the properties of this operator except for noting that it distributes over all previously defined operators except for synchronous composition.

2.6 Definition of CRE's

A concurrent regular expression is any expression consisting of symbols from a finite set Σ and +, ., *, [], ||, α, $\sigma()$ with certain constraints as summarized by the following definition.

- Any a that belongs to Σ is a regular expression (r.e.) defined over {a}. A special symbol called ϵ is also a regular expression defined over any set Σ. If A and B are r.e.'s, then so are A.B (concatenation), A+B (or), A^* (Kleene closure).
- A regular expression is also a *unit* expression. If A and B are unit expressions then so are $A||B$ (Interleaving) and A^α (Indefinite Interleaving closure).
- A unit expression is also a concurrent regular expression (cre). If A and B are cre's then so are $A||B$, $A[]B$ (synchronous composition), and $A < \sigma >$(renaming).

The intuitive idea behind above definition is as follows. We assume that a system has an infinite number of agents. Each agent is considered to have a finite number of states and therefore can be modeled by a regular set. These agents can execute independently (|| and α) and a *unit expression* models a group of agents (possibly infinite) which do not interact with each other. The world is assumed to contain a

finite number of these units which either execute independently (||) or interact by means of synchronous composition ([]).

3 Modeling of Concurrent Systems

In this section, we give some examples of use of concurrent regular examples in modeling concurrent systems.

Example 1: Producer Consumer Problem

This problem concerns shared data. The producer produces items which are kept in a buffer. The consumer takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. The problem can be specified in concurrent regular expressions as follows:
producer ::$(produce\ putitem)^*$
consumer :: $(getitem\ consume)^*$
buffer :: $(putitem\ getitem)^\alpha$
system :: producer [] buffer [] consumer

The buffer process ensures that the number of *getitem* is always less than or equal to the number of *putitem*. Note that if *alpha* is replaced by $*$ in the description of the buffer, the system will allow at most one outstanding *putitem*.

Example 2: Mutual Exclusion Problem

The mutual exclusion problem requires that at most one process be executing in the region called *critical*. It is specified in cre's as follows:
contender :: $(noncrit\ req\ crit\ exit)$
constraint :: $(req\ crit\ exit)^*$
system :: $contender^\alpha[]constraint$

Example 3: Ball Room Problem

Consider a dance ball room where both men and women enter, dance and exit. Their entry and exit need not be synchronized but it takes a pair to dance. Also we would like to ensure that the number of women in the room is always greater than or equal to the number of men, since idle men are dangerous! This system can easily be represented using a concurrent regular expression:

A man's actions can be represented by the following sequence:

man :: $menter\ dance\ mexit$

A woman's actions as follows:

woman :: $wenter\ dance\ wexit$

The constraint that the number of women always be greater can be expressed as:

constraint :: $(wenter\ (menter\ mexit) *\ wexit)^\alpha$

Since any number of men and women can enter and exit independently (except for the constraint) the entire system is modeled as follows:

man^α [] $woman^\alpha$ [] $constraint$

Example 4: $(abc)^\alpha$ [] $a^*b^*c^*$ accepts language $\{a^nb^nc^n | n \geq 0\}$. Note how the use of α operator let us keep track of number of a's that have been seen in the string. This example shows the strings that can not be recognized even by push down automata can be represented by cre's.

Example 5: Machine Shop Problem

Consider the problem of modeling a simple machine shop. The machine shop may have three machines - M_1, M_2 and M_3. It may have two operators F_1 and F_2. An order needs two stages of machining. First, it must be machined by M_1 and then by either M_2 or M_3. F_1 can operate M_1 and M_2 while F_2 can operate M_1 and M_3. Figure 1 shows the modeling by a Petri net which is not a modular description. Following is its description in concurrent regular expressions where each process is specified independently. This means that it is easier to understand and write specifications in the cre model. It is also easier to specify a partially developed system.

; b_{ij} represents the beginning of machining by i^{th} operator on j^{th} machine
; e_{ij} represents the end of machining by i^{th} operator on j^{th} machine

$order :: arrives\ phase_1\ phase_2\ leaves$
$phase_1 :: (b_{11}.e_{11}) + (b_{21}.e_{21})$
$phase_2 :: (b_{12}.e_{12}) + (b_{23}.e_{23})$
$F_1 :: (b_{11}.e_{11}) + (b_{12}.e_{12})$
$F_2 :: (b_{21}.e_{21}) + (b_{23}.e_{23})$
$M_1 :: (b_{11}.e_{11}) + (b_{21}.e_{21})$
$M_2 :: b_{12}.e_{12}$
$M_3 :: b_{23}.e_{23}$
$system :: order^{\alpha}[](F_1^*||F_2^*)[](M_1^*||M_2^*||M_3^*)$

4 Interleaving-closed sets

All the above examples include a component of the system which uses *alpha* to model an infinite number of states. Thus, α is the basic operator which increases the complexity of a system. The application of α on a set makes it interleaving-closed (i-closed). We next define i-closed sets and study their properties.

Definition: A set **A** is called *closed under repeated interleaving*, or simply *i-closed*, if for any two strings s_1 and s_2 (not necessarily distinct) that belong to **A**, $s_1||s_2$ is a subset of **A**. By definition ϵ must also belong to an i-closed set.

Examples: $\{\epsilon\}$, $\{\epsilon, a, a^2, a^3..\}$, $\{s|\#(a,s) = \#(b,s)\}$ are example of i-closed sets. As Kleene closure of a set A is the smallest set containing A and closed under concatenation, alpha closure of a set A is the smallest set containing A and closed under interleaving. More formally,

Lemma 2: Let A be a set of strings. Let B be the smallest *i-closed* set containing A. Then B = A^α.

Proof: A^α contains A and is also i-closed. Since B is smallest set with this property, we get $B \subseteq A^\alpha$.

Since B is i-closed and it contains A, it must also contain $A^{(i)}$ for all i. This implies that B contains A^α. Combining with our earlier argument we get B = A^α. Δ

The above Lemma tells us that as Kleene closure captures the notion of doing some action any number of times in series, alpha closure captures the notion of *doing some action any number of times in parallel.* Note that if a set A is i-closed, it is also concatenation closed. This is because if s_1 and s_2 belong to A then so does $s_1||s_2$, and in particular $s_1.s_2$.

We leave it to readers to verify that another definition of alpha-closure of a language A can be given as the least solution of the equation
X = (A||X) + ϵ.

Clearly taking interleaving-closure of an already i-closed set does not change it. This is formalized as follows:

Corollary: A set A is i-closed if and only if A = A^α.

Proof: If A is i-closed, it is also the smallest set containing A and i-closed. By Lemma 2, it follows that A = A^α.

Conversely, A = A^α and A^α is i-closed therefore A is also i-closed. Δ

The above corollary tells us that if a set is i-closed, then its alpha closure is the same as itself. As an application of this corollary, we get $A^{\alpha\alpha} = A^\alpha$.

4.1 Properties of I-closed Sets

As we mentioned earlier, an i-closed system may be desirable because of its logical ability to handle an infinite number of customers. Assuming that we already have designed some systems that are i-closed, we would like to combine these systems to form bigger i-closed systems. This section shows that such systems may be combined using || and [] operators but not + or . operators. All the unary operators defined in this paper preserve the property of interleaving closure. These observations are formalized in Theorems 1 and 2.

Theorem 1: If A and B are i-closed then so are A || B,A*,A^α, A[] B, and $\sigma(A)$.

Proof:

1) **A || B**:
Let s_1 and s_2 belong to A || B. We will show that $s_1||s_2$ is a subset of A || B.
$s_1 \in p_1||q_1$ because s_1 belongs to A || B , for some $p_1 \in A, q_1 \in B$.
$s_2 \in p_2||q_2$ because s_2 belongs to A || B , for some $p_2 \in A, q_2 \in B$.
therefore $s_1||s_2 \subseteq p_1||q_1||p_2||q_2$
$= p_1||p_2||q_1||q_2$ (|| is associative and commutative)
$= p||q$ *where* $p = p_1||p_2$ and q $= q_1||q_2$
$\subseteq A||$ B (because $p \subseteq A$ and $q \subseteq B$ as A and B are i-closed)

2) **A***:
As A is i-closed it is also concatenation closed and therefore $A^* = A$.

3) $\mathbf{A}^{\alpha}$: from Corollary of Lemma 2.

4) **A[]B**
Let s_1 and s_2 belong to A[]B. Then,
$s_1/\Sigma_A \in A$ and $s_1/\Sigma_B \in B$.
Similarly, $s_2/\Sigma_A \in A$ and $s_2/\Sigma_B \in B$
We will show that $s_1||s_2/\Sigma_A \subseteq A$ and $s_1||s_2/\Sigma_B \subseteq B$.
Left hand side $= s_1||s_2/\Sigma_A = s_1/\Sigma_A||s_2/\Sigma_A$ (Restriction distributes over ||)
$\subseteq A$ (A is i-closed) and similarly, $s_1||s_2/\Sigma_B = s_1/\Sigma_B||s_2/\Sigma_B \subseteq B$
Therefore, $s_1||s_2 \subseteq A[]B$.

5) $\sigma(\mathbf{A})$ Let s_1 and s_2 belong to $\sigma(A)$. Then there exists t_1 and t_2 such that $\sigma(t_1) = s_1$ and $\sigma(t_2) = s_2$, and $t_1, t_2 \in A$.
Then, $t_1||t_2 \in A$ (A is i-closed) and $\sigma(t_1||t_2) \in \sigma(A)$
This implies that $\sigma(t_1)||\sigma(t_2) \in \sigma(A)$.
Δ

For example, let $Cust_A$ and $Cust_B$ be sets of strings denoting behavior of customers in supermarket A and B respectively. Both $Cust_A$ and $Cust_B$ are i-closed and therefore, by Theorem 1, $Cust_A||Cust_B$ is also i-closed. For another example, consider the set of strings denoting the behavior of infinite customers at a supermarket. That is, Pop $= \{enter.buy.leave\}^{\alpha}$. Now assume that for buying an item a customer has to interact with the sales clerk whose behavior can be written as Clerk $= \{buy\}*$. From Theorem 1, we conclude that Pop [] Clerk is an i-closed set. However, if *buy* is not a primitive action and can be viewed as *purchase.pack* then the system is no more i-closed. It can be made i-closed as follows:
Pop :: $\{enter.purchase.pack.leave\}^{\alpha}$
Clerk1 :: $(purchase)^*$
Clerk2 :: $(pack)^*$
System :: Pop [] Clerk1 [] Clerk2

Since each of the component in the system is i-closed, by Theorem 1 the entire system is i-closed. We now show that + and . do not preserve the property of i-closure.

Theorem 2: If A and B are i-closed then A+B and A.B may not be so.

Proof:
1) **A+B**: Consider A $= \{ab\}^{\alpha}$, B$=\{bc\}^{\alpha}$. Let $s_1 = ab$ and $s_2 = bc$. Both s_1 and s_2 are

members of A+B but $s = abbc \in s_1||s_2$ does not belong to A+B.
2) **A.B**: Consider A = $\{ab\}^\alpha, B = \{bc\}^\alpha$. Let $s_1 = abbc$ and $s_2 = abbc$ Both s_1 and s_2 are members of A.B but $s = abbcabbc \in s_1||s_2$ does not belong to A.B.
Δ

5 Comparison with Other Classes of Languages

From the definition of concurrent regular expressions, we derive two new classes of languages - unit languages and concurrent regular languages. A language is called a unit language if a unit expression can describe it. Concurrent regular languages are similarly defined. In this section we study both the classes and their relationship with other classes of languages such as regular, context-free and Petri net languages.

Unit languages strictly contain regular languages and are strictly contained in Petri net languages. These languages are useful for capturing behavior of independent finite state agents which may potentially be from an infinite population. An application of such languages is the description of logical behavior of a queueing network. For example, Figure 2 shows a queueing network and a unit expression that describes the language of logical behavior of customers in it. we first show that any unit expression can be converted to a canonical form called normalized unit expression. **Lemma 3:** Let A and B be two regular expressions, then
(a) $A^\alpha||B^\alpha = (A + B)^\alpha$
(b) $(A||B^\alpha)^\alpha = A^\alpha||B^\alpha$
Proof:
(a) Let string $s \in A^\alpha||B^\alpha$
$=> s \in a_1||a_2||..||a_n||b_1||b_2||..||b_m$
for $a_i \in A, i = 1..n$, $b_j \in B, j = 1..m \quad n, m \geq 0$
$\subseteq (A + B)^\alpha$ (because each string belongs to A+B)
Let string $s \in (A + B)^\alpha$.
$=> s \in c_1||c_2||..||c_n$, where $c_i \in A + B$
If $c_i \in A$ we call it a_i, otherwise we call it b_i.
On rearranging terms so that all strings that belong to A come before strings that do not belong to A (and therefore must belong to B), we get $s \in A^\alpha||B^\alpha$.

(b) $(A||B^\alpha)^\alpha = A^\alpha||B^\alpha$

We first show that $s \in (A||B^{\alpha})^{\alpha} => s \in A^{\alpha}||B^{\alpha}$.
Let $s \in (A||B^{\alpha})^{\alpha}$
$=> s \in s_1||s_2||s_3..s_m$ where $m \geq 0$ and each $s_i \subseteq (a_i||b_{i,1}||b_{i,2}..||b_{i,n_i})$
where $b_{i,j} \in B$ for $i = 1...m$ and $j = 1...n_i$
Since || is commutative and associative all strings from set A can be moved to left and therefore s also belongs to $A^{\alpha}||B^{\alpha}$

We now show that $s \in A^{\alpha}||B^{\alpha} => s \in (A||B^{\alpha})^{\alpha}$
Let $s \in A^{\alpha}||B^{\alpha}$
$=> s \in a_1||a_2..||a_m||b_1||..||b_n$
$where\ m, n \geq 0$ and a_i's and b_i's belong to A and B respectively.
$=> s \in (a_1||\epsilon)||(a_2||\epsilon)||...||(a_{m-1}||\epsilon)||(a_m||b_1||b_2||..||b_n)$
$=> s \in (A||B^{\alpha})^{\alpha}$
Δ

Theorem 3: Any unit expression U is equivalent to another unit expression which is the interleaving of a regular expression and $(regular\ expression)^{\alpha}$. Expressions of these forms are called *normalized unit expressions.*

Proof: To show this Theorem, we use induction on the number of times || or α occurs in a unit expression. The Lemma is clearly true when the expression does not have any occurrence of || or α as a regular expression is always normalized. Assume that the Theorem holds for unit expressions with at most $k-1$ occurrences of || or α. Let U be a expression with at most k occurrences of || or α. Then U can be written as $U_1||U_2$ or U_1^{α} where U_1 and U_2 can be normalized by the induction hypothesis. We will show that U can also be normalized.
(1) $U = U_1||U_2$
$U_1 = A_1||B_1^{\alpha}$ and $U_2 = A_2||B_2^{\alpha}$
where A_1, A_2, B_1 and B_2 are regular expressions.
Therefore, $U_1||U_2 = (A_1||B_1^{\alpha})||(A_2||B_2^{\alpha})$
$= (A_1||A_2)||(B_1^{\alpha}||B_2^{\alpha})$ (|| is associative and commutative)
$= (A_1||A_2)||(B_1 + B_2)^{\alpha}$ (by Lemma 3(a))
therefore, U can be normalized.
(2) $U = U_1^{\alpha}$
$U = U_1^{\alpha} = (A||B^{\alpha})^{\alpha}$
where A and B are some regular expressions.
$U = A^{\alpha}||B^{\alpha}$ (by Lemma 3(b))
$= (A + B)^{\alpha}$ (by Lemma 3(a))
$= C^{\alpha}$ for some regular expression C.
therefore, U can be normalized.

We are now ready to explore the structure of unit languages.

Lemma 4: The unit languages properly contains the regular languages.

Proof: The containment is obvious. To see that the inclusion is proper, consider the language $(a.b)^{\alpha}$ which cannot be accepted by a finite state machine. Δ

All unit languages are also concurrent regular languages. We next show that this containment is also proper.

Definition: A language is called *i-open* if there does not exist any non-null string s such that if t belongs to a language then so does $s||t$.

Example: All finite languages are i-open. $a^*, (a+b)^*, (ab)^{\alpha}$ are not i-open because a, aba, and ab are strings respectively such that their interleaving with any string in

the language keeps it in the language. Recall that i-closed languages are set of strings that are closed under interleaving. All i-closed languages are not i-open and all i-open languages are not i-closed. However, there are languages that are neither i-open nor i-closed. An example is $a^*b^*||c^*$ which is not i-open as any interleaving with c keeps a string in the language. It is not i-closed because $abc||abc$ does not belong to the language.

Theorem 4: A unit expression cannot describe a non-regular i-open language.

Proof: Let L be a non-regular i-open language. Assume if possible that a unit expression U describes L. By Theorem 3, U can be normalized to the form $A||B^\alpha$. Since L is non-regular, the unit expression must contain at least one application of alpha-closure and therefore B is non-empty. The resulting set is not i-open as it is closed under interleaving with respect to any string in B, a contradiction.
Δ

For example, consider the language $\{a^n b^n c^n | n \geq 0\}$. The language is i-open because there is no non-null string, such that its indefinite interleaving exists in the language. By Theorem 4, we cannot construct a unit expression to accept this language. This language is concurrent regular as shown by Example 4.

Now we show that there exists i-closed languages which cannot be recognized by a single unit.

Theorem 5: There are i-closed concurrent regular languages that cannot be accepted by a unit.

Proof: Consider the concurrent regular language L = $(a_1 b_1)^\alpha [] (a_2 a_1^* b_2)^\alpha$. Assume if possible that it can be characterized by a unit expression U. By Theorem 3, U can be written as $A||B^\alpha$. Since L is an i-closed language due to Theorem 1, U is also i-closed. This implies that the language described by U is the same as that described by U^α (Lemma 2). Using Theorem 3, U can be written as C^α where C is a regular language. We will show that no such regular set exists.

Note that L contains strings starting with a_2 only. This implies that C also contains string starting with a_2 only. Further any string in L containing a single a_2 must belong to C because such a string cannot be an interleaving of two or more strings in C. Therefore, C contains all strings of the form $a_2 a_1^n b_1^n b_2$ but not $a_2 a_1^{n+k} b_1^n b_2$ for any $k > 0$. This implies that C is not a regular set, a contradiction. Δ

5.1 Relationship with Petri nets

Definition: A *Petri net* N is defined as a five-tuple (P, T, I, O, μ_0), where:
• P is a finite set of places;
• T is a finite set of transitions such that $P \cap T = \phi$
• I:T $\longrightarrow P^\infty$ is the *input* function, a mapping from transition to bag of places
• $O : T \longrightarrow P^\infty$ is the *output* function, a mapping from transition to bag of places
• μ_0 is the initial net marking, is a function from the set of places to the nonnegative integers N, $\mu_0 : P \longrightarrow N$.

Definition: A transition $t_j \in T$ in a Petri net N = (P, T, I, O, μ) is *enabled* if for all $p_i \in P$, $\mu(p_i) \geq \#(p_i, I(t_j))$ where $\#(p_i, I(t_j))$ represents multiplicity of the place p_i in the bag $I(t_j)$.

Definition: The next-state function $\delta : Z_+^n \times T \longrightarrow Z_+^n$ for a Petri net N = (P, T, I, O, μ) , $|P| = n$, with transition $t_j \in T$ is defined iff t_j is enabled. The next-state is equal to μ' where:
$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \forall p_i \in P$.
We can extend this function to a sequence of transitions as follows:
$\delta(\mu, t_j\sigma) = \delta(\delta(\mu, t_j), \sigma), \delta(\mu, \lambda) = \mu$ where λ represents the null sequence.

To define the language of a Petri net, we associate a set of symbols called alphabet Σ with a Petri net by means of a labeling function, $\sigma : T \longrightarrow \Sigma$. A sequence of transition firings can be represented as a string of labels. Let $F \subseteq P$ designate a particular subset of places as *final* places and we call a configuration μ final if
$\mu(p_i) = 0 \quad \forall p_i \in P - F$
That is, all tokens are in final places in a final configuration. If a sequence of transition firings takes the Petri Net from its initial configuration to a final configuration, the string formed by the sequence of labels of these transitions is said to be accepted by the Petri Net. The set of all strings accepted by a Petri Net is called the language of the Petri Net.
Definition: The *language* L of a Petri net N=(P, T, I, O, μ) with alphabet Σ, labeling function σ and the set of final places F, is defined as

$$L = \{\sigma(\beta) \in \Sigma^* | \beta \in T^* \; and \; \mu_f = \delta(\mu_0, \beta) \; such \; that \; \mu_f(p) = 0 \; \forall p \in P - F\}$$

Now we state the following Theorem. Since this Theorem is proved in [Garg 88] and its proof is long we provide just its sketch.
Theorem 6: The family of Petri net languages and concurrent regular languages is the same.
Sketch of the Proof: We first show that every concurrent regular expression can be converted to a Petri net. Every regular expression can be converted to a finite state machine which can be converted to a Petri net by putting a token in its start state and considering states as places. To model an infinite number of finite state agents, we can delete the start place and thus making the transitions that originated from the start place source of tokens. A Petri net corresponding to a unit expression would be interleaving of such structures. To derive a Petri net for concurrent regular expressions it is sufficient to note that Petri net languages are closed under interleaving, synchronous composition and renaming.

To convert a Petri net into concurrent regular expression, we first convert it to an ordinary Petri net using Hack's construction[Hack 75]. We then partition its places into various units. These units share transitions and have the property that for every transition, a unit has at most one place incident to it as input and one place as output. This decomposition is always possible by keeping all places in different units. We now show that a unit can be converted to a unit expression. The final concurrent regular expression is just synchronous composition of unit expressions. Since for each transition there is at most one place as input and at most one place as output, a unit can be viewed as pure interleaving of agents with their current state represented by placement of tokens. If tokens can be generated dynamically in a unit, it is modeled using alpha closure. Since there are finite number of places in a unit each agent can be viewed as a finite state machine. A finite state machine can be converted to a regular expression and therefore a unit expression would just be interleaving of regular expressions and

alpha closure of regular expressions.

For the rigorous proof of this Theorem we refer the interested readers to [Garg 88]. Figure 3 shows an example of conversion.

From above discussion, we note that

$finite \subset regular \subset unit \subset concurrent\ regular = Petri\ Net\ languages$

6 Conclusions

In this paper, we have defined an extension of regular expressions called concurrent regular expressions which can shown to be equivalent to Petri nets. The concurrent regular expression is built of regular expressions and operators - interleaving, alpha closure, synchronous composition and renaming. We have also shown the relationship of concurrent regular languages with regular, context-free and Petri net languages. Our framework supports two two main ideas. First, it provides a common framework for algebra-based and transition based models. Secondly, it supports a hierarchy of models with varying expressive power.

7 References

[**Aggarwal 87**] S.Aggarwal, D. Barbara, K.Z. Meth, "SPANNER: A Tool for Specification, Analysis, and Evaluation of Protocols", IEEE Transactions on Software Engineering, Vol 13, 12 December 1987, pp 1218-1237.

[**Ada 83**] Reference Manual for the Ada Programming Language, United States DoD, Washington, ANSI/MIL-STD-1815A-1983, 1983.

[**Billington 88**] J.Billington,G.R.Wheeler, M.C.Wilbur-Ham, "PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocols", IEEE Transactions on Software Engineering, Vol 14, 3 March 1988, pp 301-316.

[**Campbell 74**] R.H.Campbell, A.N.Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, vol 16, Springer Verlag, New York 1974, pp 89-102.

[**Campbell 79**] R.H.Campbell, R.B.Kolstad, "Path Expressions in Pascal", Proc. 4th

International Conference on Software Engineering, Munich, IEEE New York, 1979, pp 212-219.
[**Cerf 72**] V.Cerf, "Multiprocessors, Semaphores, and a Graph model of Computation," Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, California, April 1972.
[**Garg 88**] V.K.Garg, "Specification and Analysis of Distributed Systems with a Large number of Processes", Ph.D. Dissertation, University of California, Berkeley, 1988.
[**Gerhart 80**] S.L.Gerhart, et al., "An Overview of Affirm: A Specification and Verification System", Proc. IFIP 80, pp 343-348, Australia, October 1980.
[**Hoare 85**] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.
[**Hopcroft 79**] J.Hopcroft and J.Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley Pub. Co., Reading.
[**Karp 68**] R.Karp, and R.Miller, "Parallel Program Schemata", RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New York (April 1968).
[**Kosaraju 82**] R.Kosaraju, "Decidability of reachability in vector addition systems", Proc. 14th Ann. ACM Symposium on Theory of Computing, 1982, pp 267-280.
[**Lauer 79**] P.E. Lauer, P.R. Torrigiani, M.W.Shields, "COSY: A System Specification Language Based on Paths and Processes", Acta Informatica 12, pp 109-158, 1979.
[**Liskov 84**] B. Liskov,"The Argus Language and System", Proc. Advanced Course on Distributed Systems - Methods and Tools for Specification, TU Munchen, Apr. 1984.
[**Mayr 84**] E.W.Mayr, "An Algorithm for the General Petri Net Reachability Problem", SIAM Journal of Comput., Vol. 13, No.3 pp 441-460, August 1984.
[**Milne 85**] G.J.Milne,"CIRCAL and the Representation of Communication, Concurrency and Time," ACM TOPLAS, 7(2), pp 270-298, April 1985.
[**Milner 80**] A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.
[**Murata 84**] T. Murata, "Modeling and Analysis of Concurrent Systems", in book Handbook of Software Engineering, ed. C.R.Vick and C.V.Ramamoorthy, Publ.Van Nostrand Reinhold, pp 39-63, 1984.
[**Peterson 81**] J. Peterson, Petri-Net Theory and Modeling of Systems, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.
[**Pratt 86**] V. Pratt, "Modeling Concurrency with Partial Orders", International Journal of Parallel Programming, Vol. 15, No. 1, February 1986, pp 33-71.
[**Reisig 85**] W. Reisig, Petri Nets, An Introduction, lecture notes in Computer Science, Springer-Verlag, 1985.
[**Taylor 83**] R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs", Communications of the ACM, 26(5) pp 362-376, 1983.
[**Inan 88**] K. Inan and P. Varaiya, "Finitely Recursive Processes for Discrete Event Systems", IEEE Transactions on Autmatic Control, 33(7):626-639, July 1988.
[**Zave 85**] P.Zave, "A Distributed Alternative to Finite-State-Machine Specifications", ACM Transactions on Programming Languages and Systems Vol 7, No 1, January 1985, pp 10-36.