Concurrent Regular Expressions and their Relationship to Petri Nets

Vijay K. Garg [1]
Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712-1084
M.T.Raghunath
Computer Science Division,
University of California, Berkeley
Berkeley, CA 94720

Abstract

We define algebraic systems called concurrent regular expressions which provide a modular description of languages of Petri nets. Concurrent regular expressions are extension of regular expressions with four operators - interleaving, interleaving closure, synchronous composition and renaming. This alternative characterization of Petri net languages gives us a flexible way of specifying concurrent systems. Concurrent regular expressions are modular and hence easier to use for specification. The proof of equivalence also provides a natural decomposition method for Petri nets.

1 Introduction

Formal models proposed for specification and analysis of concurrent systems can be categorized roughly into two groups: *algebra based* and *transition based.* The algebra based models specify all possible behaviors of concurrent systems by means of expressions that consist of algebraic operators and primitive behaviors. Examples of such models are path expressions[3], behavior expressions[21] and extended regular expressions. Examples of tools to analyze the specifications based on such models are Path Pascal[4], COSY [17], CCS [21] and Paisley [30]. The transition based models provide a computational model in which the behavior of the system is generally modeled as a configuration of an automaton from which one or more transitions are possible. Examples of the transition based models are finite state machines[12], S/R Model[1], UCLA graphs[5], and Petri nets[27]. Examples of modeling and analysis tools based on these models are Spanner [1], Affirm [9] and PROTEAN [2].

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,

[1]supported in part by grants from the Bureau of Engineering Research and University Research Institute, University of Texas at Austin

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.

The languages of Petri nets have also been studied by [10, 23, 26, 28, 29]. Hack [10] and Peterson [23] studied closure properties of Petri net languages but did not provide any characterization of their languages. [26] provides a characterization in terms of Szilard languages of matrix context-free languages. Our characterization is much simpler and provides a clear relationship between regular sets and Petri net languages. Moreover, it uses operators that arise naturally in modeling concurrent systems such as interleaving and synchronous composition.

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. Analysis questions such as reachability are more computationally expensive for Petri nets than for finite state machines. 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[21], CSP[11] and FRP[13] These models do not have any equivalent transition based model. Similarly, they do not support a hierarchy of models like we do. Path expressions[17] were shown to be translatable to Petri nets, and thus analyzable for reachability properties [14, 15, 19]. Concurrent regular expressions are more general than Path expressions as they are equivalent to Petri nets.

We have used interleaving semantics rather than true concurrency as advocated by [25] and [27]. This assumption is in agreement with CSP[11] and CCS[21]. 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[11], and synchronization trees[21], 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[11].

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 concurrent regular expressions for modeling distributed systems. Section 4 compares the class of languages defined by concurrent regular expressions with regular, 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 describe 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
$L_1 + L_2 = L_1 \cup L_2$ 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 [12].

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) A || B = B || A   *(Commutativity)*
(2) A || (B || C) = (A || B) || C   *(Associativity)*
(3) A $||\{\epsilon\}$ = A   *(identity of ||)*
(4) A $||\phi = \phi$ *(zero of ||)*
(5) (A+B) || C = (A || C)+(B || C)   *(Distributivity over +)*

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

Lemma 0: 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[12]. 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 the symbol a in the string w, the interpretation of $CUST^\alpha$ is as follows:
$CUST^\alpha = \{w| \ for \ all \quad prefixes \ s \ of \ w, \#(enter, s) \geq \#(buy, s) \geq \#(leave, s)$, 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 [12]. 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 $= \phi$ 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 rename the event symbols of a process. Some examples are:

- 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:

$\sigma(L_1) = \{\sigma(s) | s \in L_1\}$

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, $\sigma()$, and ϵ with certain constraints as summarized by the following definition.

- Any a that belongs to Σ is a regular expression (r.e.). A special symbol called ϵ is also a regular expression. 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 $\sigma(A)$(renaming).

The intuitive idea behind above definition is as follows. We assume that a system has multiple (possibly infinite) agents. Each agent is assumed 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 α 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:

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

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.

4 Relationship with Petri Nets

In this section, we show that concurrent regular expressions characterize the class of Petri Net languages. The proof of this characterization involves following steps.

1. We define an automata theoretic model called Decomposed Petri Nets (DPN). We show that any Petri net can be converted to a Decomposed Petri Net such that they have the same language. A DPN consists of one or more units. The decomposition involves partitioning of places of the original Petri net into various units such that each unit models a set of non-interacting processes.

2. We show how a DPN can be converted to concurrent regular expressions. Intuitively, each unit consists of interleaving of finite state processes (possibly an infinite number of them) each of which could be characterized by a regular expression.

3. We show how any concurrent regular expression can be converted to a Petri net such that they have the same language. This transformation uses various closure properties of Petri net languages.

Thus a system can be expressed in Petri net, DPN, or CRE formalism and transformed to any other formalism. This transformation can be used for systems which are easier to specify in one formalism but easier to analyze in other.

The above proof provides a new decomposition method for Petri nets. This method has the advantage of separating concurrency and synchronization in Petri nets. The resulting automata called Decomposed Petri Net and their equivalent concurrent regular expressions satisfy *modularity* properties and can be more easily used for specification of concurrent systems.

4.1 Languages of 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 $\mathcal{N}$, $\mu_0 : P \longrightarrow \mathcal{N}$.

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:
$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \qquad for\ all\ p_i \in P.$
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

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

Note that our notion of final configurations is different from the traditional definition of Petri net languages which typically use a *finite* set of final configurations (cf. [Peterson 83]). Our definition of final configurations may result in infinite number of them. Our results provide a strong motivation for using our definition of final configurations.

4.2 Transformation of PN's to DPN's

As we said earlier, it is convenient to decompose a given Petri net for the purposes of our characterization. A Petri net is partitioned into multiple *units* which share all the transitions of the Petri net. Each unit contains some of the places of the original Petri net. Intuitively, the decomposition is such that the tokens within a unit need to synchronize only with tokens in other units. Each unit is a generalization of finite state machine. Formally, a DPN (Decomposed Petri Net) D is a tuple (T, U) where

- T = a finite set of symbols called *transition alphabet*
- U = set of units $(U_1, U_2..U_n)$ where each unit is a five tuple i.e. $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ where:
  - P_i is a finite set of *places*
  - C_i is an initial *configuration* which is a function from the set of places to nonnegative integers $\mathcal{N}$ and a special symbol '*'. i.e.,$C_i : P_i - > (\mathcal{N} \bigcup \{*\})$. The symbol '*' represents an unbounded number of tokens. A place which has * tokens is called a **-place*.
  - Σ_i is a finite set of *transition* labels s.t. $\Sigma_i \subseteq T$.
  - δ_i is a relation between $P_i \times \Sigma_i$ and P_i, i.e., $\delta_i \subseteq (P_i \times \Sigma_i) \times P_i$. δ_i represents all transition arcs in the unit.
  - F_i is a set of final places, $F_i \subseteq P_i$.

The configuration of a DPN can change when a transition is fired. A transition with label a is said to be *enabled* if for all units $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ such that $a \in \Sigma_i$ there exists a transition (p_k, a, p_l) with $C_i(p_k) \geq 1$. Informally, a transition a is enabled if all the units that have a transition labeled a, have at least one place with non-zero tokens and an outgoing edge labeled a. For example, in Figure 1 *get-item* is enabled only if both p_4 and p_5 have tokens. A transition may *fire* if it is enabled. The firing will result in a new marking C'_i for all participating units, and is defined by
$C'_i(p_k) = C_i(p_k) - 1$
$C'_i(p_l) = C_i(p_l) + 1$.
A *-place remains the same after addition or deletion of tokens.

As an example of a DPN machine, consider the producer consumer problem. 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 consumer can execute *get-item* only if there is a token in the place p_4. Note how the *-place is used to represent an unbounded number of tokens.

Figure 1: A DPN machine for Producer Consumer Problem

The definition of the language of a DPN is identical to that of a PN.

Theorem 1: *Every Petri net can be decomposed, i.e., for every PN there exists a DPN such that they have the same language.*

Before we prove this result, we will need the following Lemma which is based on a result by Hack [10].

Lemma 1 : For every Petri net P, there exists an ordinary Petri net such that they have the same language.

Proof: We can use a construction provided by [10] to convert any Petri net to an ordinary Petri net such that its language is preserved. This construction replaces a place with maximum multiplicity of k by a ring of k places each having multiplicity of 1. The tokens can move freely within this ring by means of ϵ labeled transition. A similar result has also been shown by [16].

Proof of Theorem 1: We will show that any ordinary Petri net can be decomposed to a DPN and then using Lemma 1 we can assert this result for any Petri net.

(1) Construction of a DPN from an Ordinary Petri net

Let N be a Petri net $= (P, T, I, O, \mu, F)$ with the usual meaning of the notation. Every place in the Petri net is also a place in the DPN. These places, however, may belong to different units depending on the *unit assignment function.* A unit assignment function is any function $f : P- > \{1, 2, ..k\}$ such that

$$\forall t \in T, p_1, p_2 \in P : \quad ((p_1, p_2) \subseteq I(t)) \vee ((p_1, p_2) \subseteq O(t)) => f(p_1) \neq f(p_2)$$

This condition implies that places belonging to the same unit cannot be input(output) to the same transition. It holds trivially if all places belong to different units.

We define the DPN D as $D = (T, \{U_1, U_2, ...U_K\})$ where $U_i = (P_i, \Sigma_i, C_i, \delta_i, F_i)$ is defined as follows:

• P_i contains all the places that are assigned the unit number i, and a *-place denoted by sp_i.

$$P_i = \{p \in P | f(p) = i\} \cup \{sp_i\}$$

• Σ_i contains as transition symbols all those transitions in which places belonging to unit i participate.

$$\Sigma_i = \{t \in T | \exists p \in P_i, p \in I(t) \cup O(t)\}$$

• The configuration of the DPN ($C_i : P_i- > N \cup \{*\}$) is the same as the marking function in the Petri net, i.e.

$$C_i(p) = \mu(p) \; \forall p \in P_i, \quad C_i(sp_i) = *.$$

• $\delta_i \subseteq P_i \times \Sigma_i \times P_i$. If a unit has an input place as well as an output place for a transition, an arc is added between them. If a unit has only an input place for a transition then an arc is added between the input place and its *-place. If a unit has only an output place for a transition then an arc is added between its *-place and the output place. Formally,

$\delta_i = \{(p_j, t, p_k) | \exists t : (p_j \in I(t)) \wedge (p_k \in O(t))\}$

$\cup \{(p_j, t, sp_i) | \exists t : p_j \in I(t), \nexists p_k, p_k \in O(t)\}$

$\cup\{(sp_i, t, p_k) | \exists t : p_k \in O(t), \nexists p_j, p_j \in I(t)\}$
• F_i is the set of final places.

$$F_i = (P_i \cap F) \cup \{sp_i\}$$

Thus, *-places are always final places.

The size of the resulting DPN is of the same order as the size of the Petri net. Also, the transformation of the given Petri net structure can be done in linear time.

The set of sequences of transitions is identical for both structures because:
(1) Initially, both the Petri net and the DPN have the same configuration.
(2) The set of transitions that is enabled for equal configurations is identical.
(3) Both machines starting from equal configurations reach equal configurations on taking the same transition. Δ

4.3 Transformation of DPN's to Concurrent Regular Expressions

We next show that there exists an algorithm to derive a concurrent regular expression that describes the set of strings accepted by a DPN. We need the following Lemmas before we can prove the required result.

Lemma 2.1: Any unit with multiple *-places can be converted to an equivalent unit with a single *-place.

Proof: Merge all *-places into a single *-place. All input arcs and output arcs in the unit are combined. Since the tokens in *-places do not change and the bag of transitions enabled for any configuration is identical, we conclude that the language remains the same. Δ

Lemma 2.2: Any unit U is equivalent to another unit which has at most two connected components - one with *-place and the other with a single token.

Proof : From Lemma 2.1, we can assume, without loss of generality, that there is at most one *-place in U. U may have one or more connected components. Let the connected component C have the *-place. C may have tokens at some non-* places too. As tokens move independently of each other within a unit, C can be written as two components- one with tokens only in the non-* places and the other with the *-place. All the connected components of U with no *-places can be combined into a single connected component - a finite state machine. This is because there is a finite number of invariant tokens residing in finite number of places, resulting in only a finite number of possible configurations. Therefore, a finite state machine can simulate the behavior of these components. Δ

Lemma 2.3: Let U be a unit with a single *-place having no tokens in its simple places. Then its language can be written as a $(regular\ expression)^{\alpha}$.

Proof: Let $U = (P, C, \Sigma, \delta, F)$ with $C(p_i) = *$. We construct the finite state machine $A = (P, p_i, \Sigma, \delta, F)$, with p_i as the initial state. Let L(X) represent the language accepted by automata X. We will show that $L(U) = L(A)^{\alpha}$.

Case 1: $L(U) \subseteq L(A)^{\alpha}$

Let a string s belong to the language of the unit U. In accepting s, a finite number of tokens, say n, must have moved from the *-place to some final place. Let $s_1, s_2..s_n$ be the strings

that are traced by tokens $1..n$, respectively, such that one of their interleaving is s. Each of the strings $s_1..s_n$ also belongs to the regular set. Therefore, their interleaving belongs to α-closure of the regular set.
Case 2: $L(A)^\alpha \subseteq L(U)$
Consider any string s in $L(A)^\alpha$. This string s can be written as $s_1||s_2||..||s_n$ where each s_i belong to A. As s_i belong to A, it also represents a path from the initial place to a final place in U. Hence s can be simulated by n tokens which simulate $s_1, ..s_n$ respectively. Δ
Theorem 2: *There exists an algorithm to derive a concurrent regular expression that describes the set of strings accepted by a DPN.*
Proof : To derive the expression for a unit, we use Lemma 2.2 to convert it into a unit with at most two components, one with *-place and one with a single token. From Lemma 2.3, the language of any such unit can be written as interleaving of a regular expression and at most one $(regular\ expression)^\alpha$. The concurrent expression equivalent to the DPN will be the unit expressions for units composed by the [] operator. We can finally apply the labeling function used for defining the Petri net's language as the renaming function. Δ

An example of equivalent Petri net, DPN and concurrent regular expression is shown in Figure 2. Note that, it is easy to show that number of *a's* in any prefix is greater than number of *c's* by considering the language of unit 2. Similarly, from unit 1 it is clear that the events b and d alternate in the system.

Figure 2: PN => DPN => CRE

4.4 Transformation of a CRE to a PN

To show that every CRE can be converted to a Petri Net, we need the following Lemmas.
Lemma 3.1: 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

Lemma 3.2: Any unit expression U is equivalent to another unit expression which is the interleaving of a regular expression and $(\textit{regular expression})^\alpha$. 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. Δ

Lemma 3.3: If L_1 and L_2 are Petri net languages defined over Σ_1 and Σ_2, then

(1) $L_1||L_2$ is a Petri net language defined over $\Sigma_1 \cup \Sigma_2$.

(2) $L_1[]L_2$ is a Petri net language defined over $\Sigma_1 \cup \Sigma_2$.

(3) $\sigma(L_1)$ is a Petri net language defined over $\sigma(\Sigma_1)$.

Proof:
Any Petri net $N = (P, T, I, O, \mu)$ with alphabet Σ, labeling σ and the set of final places F can be converted to a Petri net which has token initially at only one place, say p_s. To do this, construct a special place called p_s, and a null labeled transition which ensures that the initial number of tokens are put after it fires. Therefore, we can construct Petri nets in standard form that accept L_1 and L_2.
(1) A new start place is defined from which a token goes to start places of both the Petri nets.
(2) At a given point in the string if a transition fires in a Perti net and its label is in $\Sigma_1 \cap \Sigma_2$, then a transition in the other Petri net with the same label must also fire. Thus, a new transition is created by combining the two transition with the same label in the two Petri nets. When more than one transition exists with the same label, all possible pairs of transitions must be considered.
(3) The new language can be generated by the old Petri net with the labeling function as $\psi.\sigma$ where ψ is the old labeling function. Δ

Theorem 3: *There exists an algorithm to derive a Petri net that describes the set of strings described by a concurrent regular expression.*

Proof: Note that a concurrent regular expression is either a unit expression or concurrent regular expressions composed with [], ||, and $\sigma()$. Since by Lemma 3.3, Petri net languages are closed under all these operators, it is sufficient to derive a Petri net for a unit expression. By Lemma 3.2 any unit expression can be converted to a unit automaton such that they accept the same language. It is easy to construct a Petri net from a unit by treating each arc label as a transition and deleting the *-places. 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.

Figure 3: A Queueing Network and its Equivalent CRE

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 3 shows a queueing network and a unit expression that describes the language of logical behavior of customers in it.

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

Theorem 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. To show this we need to define i-closed and i-open sets.

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.

Some examples of i-closed sets are: $\{\epsilon\}$, $\{\epsilon, a, a^2, a^3..\}$, $\{s | \#(a,s) = \#(b,s)\}$. 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 5.1: 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 5.1, 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 $A^{\alpha\alpha} = A^\alpha$.
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 5: *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 Lemma 3.2, 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 $\{a^nb^nc^n|n \geq 0\}$. The language is i-open because there is no non-null string, such that its indefinite interleaving exists in the language. By Theorem 5, we cannot construct a unit expression to accept this language. This language is concurrent regular as shown by Example 4 in Section 3.

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

Theorem 6: *There are i-closed concurrent regular languages that cannot be accepted by a unit.*

Proof: Consider the concurrent regular language L = $(a_1b_1)^\alpha[](a_2a_1^*b_2)^\alpha$. Assume if possible that it can be characterized by a unit expression U. Since L is an i-closed language U is also i-closed. This implies that the language described by U is the same as that described by U^α (Lemma 5.1). Using Lemma 3.2, 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 regular.

From the above discussion, we conclude that
regular languages $\subset$ *unit languages* $\subset$ *concurrent regular languages*

6 Conclusions

This paper makes two contributions to Petri Net theory. First, it provides an alternative description of Petri net languages. This description is in terms of natural operators such as interleaving and synchronization. Based on this description it is easier to understand the behavior of systems modeled by Petri nets.

Secondly, it provides a decomposition of Petri nets. The resulting model, DPN possesses modular properties. Each module or unit defines a set of non-interacting processes and therefore can be modeled and studied in isolation from rest of the system. Similarly, DPN's have a closer correspondence with state machines and since the notion of state arises in many contexts, they are easier to use for specification and analysis of concurrent systems. Applications of DPN for specification of concurrent systems are shown in [6,7]. Concurrent regular expressions are used for modeling synchronization constraints in the language ConC [8].

7 References

[1] 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.
[2] 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.
[3] 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.
[4] 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.
[5] V.Cerf, "Multiprocessors, Semaphores, and a Graph model of Computation," Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, California, April 1972.
[6] V.K.Garg, "Specification and Analysis of Distributed Systems with a Large number of Processes", Ph.D. Dissertation, University of California, Berkeley, 1988.

[7] V. Garg, "Modeling of Distributed Systems by Concurrent Regular Expressions", *Proc. 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols,* Vancouver, Dec 1989.
[8] V. Garg, C.V. Ramamoorthy, "ConC: A Language for Concurrent Programming", *accepted for* Computer Languages Journal.
[9] S.L.Gerhart, et al., "An Overview of Affirm: A Specification and Verification System", Proc. IFIP 80, pp 343-348, Australia, October 1980.
[10] M. Hack, "Petri Net languages," *Computation Structures Group Memo 124,* Project Mac, Massachusetts Institute of Technology , June, 1975.
[11] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.
[12] J.Hopcroft and J.Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley Pub. Co., Reading.
[13] K. Inan and P. Varaiya, "Finitely Recursive Processes for Discrete Event Systems", IEEE Transactions on Automatic Control, 33(7):626-639, July 1988.
[14] R.Karp, and R.Miller, "Parallel Program Schemata", RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New York (April 1968).
[15] R.Kosaraju, "Decidability of reachability in vector addition systems", Proc. 14th Ann. ACM Symposium on Theory of Computing, 1982, pp 267-280.
[16] S. Lafortune,"On Petri Net Languages", EECS Department, University of Michigan at Ann Arbor.
[17] 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.
[18] B. Liskov,"The Argus Language and System", Proc. Advanced Course on Distributed Systems - Methods and Tools for Specification, TU Munchen, Apr. 1984.
[19] 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.
[20] G.J.Milne,"CIRCAL and the Representation of Communication, Concurrency and Time," ACM TOPLAS, 7(2), pp 270-298, April 1985.
[21] A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.
[22] 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.
[23] J. Peterson, "Computation Sequence Sets," *Journal of Computer and System Sciences*, Volume 13, Number 1, 1976, pp 1-24.
[24] J. Peterson, Petri-Net Theory and Modeling of Systems, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.
[25] V. Pratt, "Modeling Concurrency with Partial Orders", International Journal of Parallel Programming, Vol. 15, No. 1, February 1986, pp 33-71.
[26] S. Crespi-Reghizzi and D. Mandrioli, "Petri Nets and Szilard Languages," *Information and Control*, Volume 33, Number 2, 1977, pp 177-192

[27] W. Reisig, Petri Nets, An Introduction, lecture notes in Computer Science, Springer-Verlag, 1985.
[28] P. Starke, "Free Petri Net Languages," *Seventh Symposium on Mathematical Foundations of Computer Science*, 1978.
[29] R. Valk and G. Vidal-Naquet, "Petri Nets and Regular Languages," *Journal of Computer and System Sciences*, Volume 23, 1981, pp 229-325.
[30] 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.