JOURNAL OF COMPUTER AND SYSTEM SCIENCES 13, 1–24 (1976)

Computation Sequence Sets

JAMES L. PETERSON

Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712

Received October 10, 1974; revised June 18, 1975

A class of automata based upon generalized Petri nets is introduced and defined. The language which a Petri net generates during an execution is called a computation sequence set (CSS). The class of CSS languages is shown to be closed under union, intersection, concatenation, and concurrency. All regular languages and all bounded context-free languages are CSS, while all CSS are context-sensitive. Not all CSS languages are context-free, nor are all context-free languages CSS. Decidability problems for CSS hinge on the emptiness problem for CSS. This problem is equivalent to the reachability problem for vector addition systems, and is open.

1. INTRODUCTION

Petri nets have been used by several researchers for the description and analysis of systems of parallel processes [8, 9, 16, 17, 18]. Although the majority of current research with Petri nets is still directed toward parallel computation, in this paper we consider Petri nets as an automaton in the same way as finite state machines, pushdown stack automata, and Turing machines. Viewed in this way, a language cen be naturally associated with the execution of a Petri net. Consideration of the properties of the class of languages generated by Petri nets yields both new properties of Petri nets and an interesting addition to formal language theory.

We first define the new class of automata based on Petri nets. Then, the language of a Petri net, called a computation sequence set (CSS), is defined. A computation sequence set contains all possible computation sequences which may represent an execution of a Petri net from its start state to a final state. Formal definitions of these concepts are given in Section 2. Section 3 investigates the closure properties of the class of computation sequence sets, and Section 4 relates this new class of languages to the classical hierarchy of regular, context-free, context-sensitive, and type-0 languages. Section 5 then considers some decidability questions and conclusions about CSS as a class of languages.

This work supported, in part, by the National Science Foundation under Grant MCS75-16425.

2. The Petri Net

We begin by giving a definition for the class of Petri nets. This definition follows the approach of [17] and is essentially the same as the Generalized Petri Nets of [6] although different notation is used. This general definition subsumes, or is equivalent to, most other definitions of Petri nets.

2.1. *Definition of the Petri Net*

A Petri net, C, is a 5-tuple defined by

$$C = (P, T, \Sigma, S, F),$$

where

$P = \{p_1, p_2, \ldots, p_n\}$ is a set of *places*,

$T = \{t_1, t_2, \ldots, t_m\}$ is a set of *transitions*,

Σ is the *input alphabet*, a set of *symbols* or *labels*,

$S \in P$ is the *start place*,

$F \subseteq P$ is the set of *final places*.

Each transition, $t_j \in T$, is an ordered triple defined by

$$t_j = (\sigma_j, I_j, O_j)$$

where

σ_j is the *symbol* or *label* associated with t_j ($\sigma_j \in \Sigma$),

I_j is the bag of *input places* for t_j ($I_j \in P^\infty$),

O_j is the bag of *output places* for t_j ($O_j \in P^\infty$).

(The Appendix gives a brief summary of the theory and notation of bags. Bags are essentially an extension of sets which allow multiple occurrences of an element in a bag. The number of occurrences of an element x in a bag β is given by the function $\#(x, \beta)$. Our use of bags is for descriptive purposes, so we use the notation and concepts of set theory with which the reader should be familiar. For a more complete development of bag theory, the reader is referred to [2].)

The sets P, T, Σ are assumed to be finite. The cardinality of the set P is n and of the set T, m. Arbitrary elements of P and T are denoted by p_k ($1 \leqslant k \leqslant n$) and t_j ($1 \leqslant j \leqslant m$), respectively. The set Σ is not generally defined explicitly since it can be inferred from the definitions of the transitions ($\Sigma = \{\sigma_j \mid (\sigma_j, I_j, O_j) \in T\}$). We use σ, σ_j and early lowercase Roman letters ($a, b, c, \ldots$) to represent elements of Σ.

An example Petri net is defined in Fig. 1.

$$C = (P, T, \Sigma, S, F)$$
$$P = \{p_1, p_2, p_3, p_4, p_5\}$$
$$T = \{t_1, t_2, t_3, t_4\}$$
$$\Sigma = \{a, b, c\}$$
$$S = p_1$$
$$F = \{p_5\}$$

$$t_1 = (a, \{p_1\}, \{p_2, p_3, p_3, p_5\})$$
$$t_2 = (b, \{p_2, p_3, p_5\}, \{p_5\})$$
$$t_3 = (c, \{p_3\}, \{p_4\})$$
$$t_4 = (c, \{p_4\}, \{p_2, p_3\})$$

FIG. 1. Definition of an example Petri net.

When working with Petri nets, we need to refer to the separate components of the ordered triples which define the transitions. To allow us to specify easily the portion of a transition which we are discussing, we define three projection functions—the label function (σ), the input function (I), and the output function (O). For a transition $t_j = (\sigma_j, I_j, O_j)$, these functions are defined by

$$\sigma(t_j) = \sigma_j,$$
$$I(t_j) = I_j,$$
$$O(t_j) = O_j.$$

To map sequences of transitions into sequences of symbols, we extend the label function by

$$\sigma(x) = \epsilon \qquad \text{if} \quad x = \epsilon,$$
$$\sigma(x) = \sigma(t_j)\,\sigma(y) \qquad \text{if} \quad x = t_j y,\ t_j \in T,\ y \in T^*.$$

(We use ϵ to denote the empty sequence. Σ^* denotes the set of all strings over an alphabet Σ.)

A convenient visual representation of a Petri net is a bipartite directed graph. Both places and transitions are represented as nodes in the graph. To distinguish them, places are represented by circles and transitions by bars. An arc is directed from a transition t_j to a place p_k for each occurrence of p_k in the output bag, $O(t_j)$, of the transition. An arc is directed from a place p_k to a transition t_j for each occurrence of p_k in the input bag, $I(t_j)$, of t_j. Since the ordering of places and transitions is unimportant, the start place is assumed to be p_1. Final places are indicated by a circle around the node representing them. The Petri net of Fig. 1 is graphed in Fig. 2.

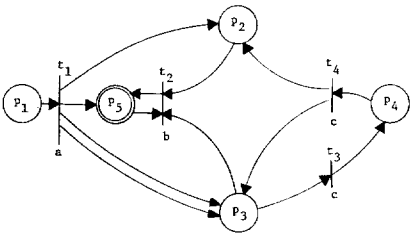


FIG. 2. Graphical representation of the Petri net of Fig. 1.

The graph representation of a Petri net contains all the information which is necessary to define the net. Thus we give graph representations of Petri nets rather than formal definitions for our illustrations.

2.2. *Execution Rules for a Petri Net*

The above definitions are concerned with the description of the structural properties of a Petri net. Since the Petri net is an abstract machine, it also has computational properties. The computational properties refer to its behavior during an execution. The execution of a Petri net is directed by the existence and location of *tokens* in the net. Tokens are abstract entities which we represent by black dots in the circles of the graphical representation of a Petri net. Tokens move about the Petri net in a manner dictated by the execution rules for Petri nets. These rules are

(1) The Petri net is initialized by placing one token (the *start token*) in the start place.

(2) If the net is in a final state, we may halt; otherwise the set of enabled transitions, U, is computed.

(3) If U is nonempty, one transition from U is fired, and we may return to step (2). If U is empty, the execution halts.

A transition is enabled if all of its input places have (a sufficient number of) tokens in them. A transition fires by removing tokens from all of its input places and placing tokens in all of its output places. These definitions are made more precise by

DEFINITION. A transition, t_j, is *enabled* if for each $p_k \in P$, there are at least $\#(p_k, I(t_j))$ tokens in p_k.

DEFINITION. An enabled transition, t_j, *fires* by first removing $\#(p_k, I(t_j))$ tokens from each $p_k \in P$, and then adding $\#(p_k, O(t_j))$ tokens to each $p_k \in P$.

Execution of a Petri net begins with one token in the start place. Each time that a transition fires, it may change the number and/or location of tokens in the Petri net and therefore the state of the net. A Petri net may halt whenever it reaches a *final state*

(one token in a final place and zero tokens elsewhere) or it may continue execution. If the set, U, of enabled transitions is empty, the Petri net must halt.

Figure 3 illustrates the concept of the execution of a Petri net by using the graphical representation of Fig. 2 to present one possible execution. At each step, the Petri net and its tokens are given as well as the set U of enabled transitions and the selected transition which fires.

2.3. *The State Space of a Petri Net*

The *state* of a Petri net is defined by the number and location of tokens in the net. This can also be expressed as the number of tokens (possibly zero) in each place of the net and is commonly called a *marking*. The number of tokens in each place will always be a nonnegative integer number, and we represent the state of a Petri net by an n-vector of nonnegative integers. The firing of a transition represents a change in the state of the Petri net. A state is *reachable* if there exists some sequence of firings which transforms the start state (the state associated with one token in the start place and zero tokens elsewhere) into the desired state.

We define Q to be the *reachable state space* of a Petri net. Q is also called the *marking*

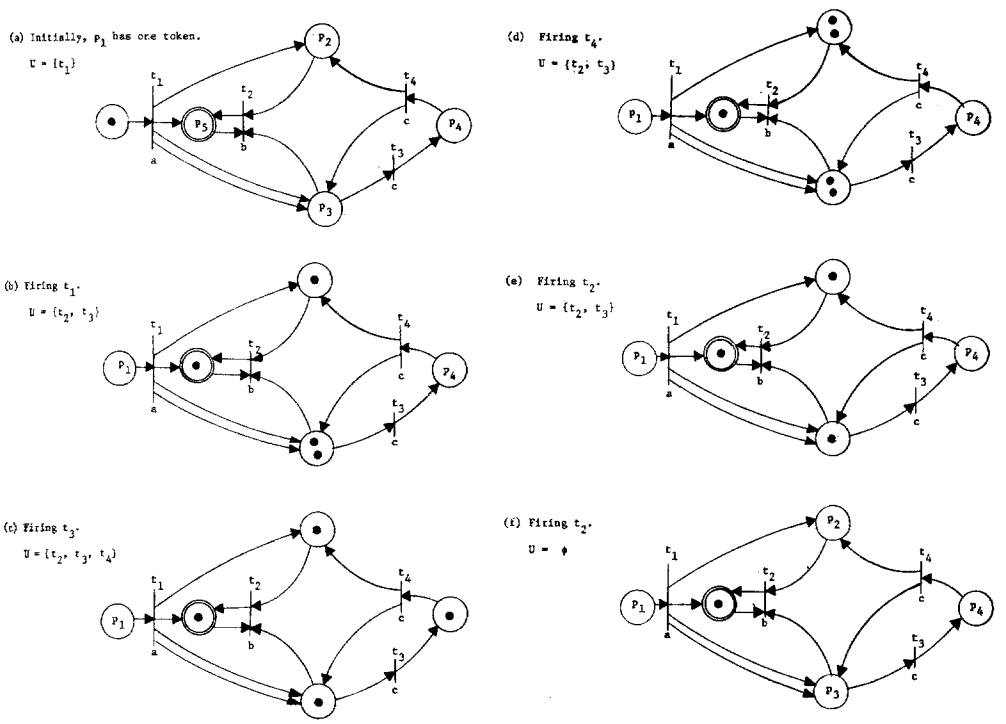


FIG. 3. One possible execution of the Petri net of Fig. 1.

class of a Petri net. If $\mathbb{N}$ represents the set of nonnegative integers then $Q \subseteq \mathbb{N}^n$. Each element of Q is an n-vector whose kth component represents the number of tokens in place p_k ($1 \leqslant k \leqslant n$). We denote by S both the start place and the vector (1, 0, 0,...); F denotes both the set of final places and the set of vectors representing one token in a final place and zero tokens elsewhere (the final states).

The next-state function, δ, is a (partial) function from $\mathbb{N}^n \times T$ into $\mathbb{N}^n$. For a state vector, q, and a transition, t_j, the next-state function, $\delta(q, t_j)$, is defined if and only if for all k, $1 \leqslant k \leqslant n$,

$$q_k \geqslant \#(p_k, I(t_j)).$$

Thus a transition t_j is enabled in a state q if and only if $\delta(q, t_j)$ is defined. If $\delta(q, t_j)$ is defined, then the new state vector defined is the state resulting from the firing of t_j. The kth component of the new state is defined by

$$\delta(q, t_j)_k = q_k - \#(p_k, I(t_j)) + \#(p_k, O(t_j)).$$

Since $q_k \geqslant \#(p_k, I(t_j))$ if $\delta(q, t_j)$ is defined and $\#(p_k, O(t_j)) \geqslant 0$, we see that if $\delta(q, t_j)$ is defined, then $\delta(q, t_j) \geqslant 0$ and hence $\delta(q, t_j) \in \mathbb{N}^n$.

The definition of $\delta(q, t_j)$ can be recast as a vector replacement system [12]. We specify, for each transition, t_j, two vectors, u_j and v_j, where $(u_j)_k = -(p_k, I(t_j))$ and $(v_j)_k = -\#(p_k, I(t_j)) + \#(p_k, O(t_j))$. Then $\delta(q, t_j)$ is defined if $q + u_j \geqslant 0$, and if $\delta(q, t_j)$ is defined, then $\delta(q, t_j) = q + v_j$. The reachable state space of a Petri net corresponds to the reachability set of a vector replacement system (see Section 5).

As with the label function, we extend the next-state function from a domain of individual transitions to a domain of sequences of transitions. If x is a sequence of transitions ($x \in T^*$), then

$$\begin{aligned} \delta(q, x) &= q && \text{if} \quad x = \epsilon, \\ &= \delta(\delta(q, t_j), y) && \text{if} \quad x = t_j y \qquad \text{for} \quad t_j \in T, \quad y \in T^*. \end{aligned}$$

Of course $\delta(q, x)$ is defined if and only if the next-state functions of the above definition are defined for their arguments.

We can now formally define the reachable state space, Q, as the smallest subset of $\mathbb{N}^n$ defined by

(a) $S \in Q$.

(b) if $q \in Q$, and $\delta(q, x)$ is defined for $x \in T^*$, then $\delta(q, x) \in Q$.

Since we are concerned only with reachable states, we restrict the next-state function to the reachable state space, Q. Thus, $\delta: Q \times T^* \to Q$, and (except perhaps for the start state) the mapping is onto.

It should be clear from the definition of the state space, the next-state function, and the reachable state space that the automaton defined by $(Q, \delta, \Sigma, S, F)$ is equivalent

to (P, T, Σ, S, F) as a mathematical formulation of a Petri net. We use both definitions interchangably.

2.4. *Transition Sequences and Computation Sequences*

Each separate execution of a Petri net defines, or is defined by, the sequence of transitions which are fired during the execution of the net. We say that a sequence of transitions, $x \in T^*$, is *legal* if it represents a possible sequence of transition firings from the start state, S. Thus a sequence is legal if $\delta(S, x)$ is defined. A sequence is *complete* if it is legal and $\delta(S, x) \in F$.

To illustrate these concepts, consider the execution shown in Fig. 3. This execution is completely defined by the transition sequence $t_1t_3t_4t_2t_2$. For this example, the sequence is both legal and complete. The sequences $t_1t_3t_4$ and $t_1t_3t_3t_4t_4t_2t_2$ are legal but not complete, since

$$\delta(S, t_1t_3t_4) = (0, 2, 2, 0, 1),$$

$$\delta(S, t_1t_3t_3t_4t_4t_2t_2) = (0, 1, 0, 0, 1).$$

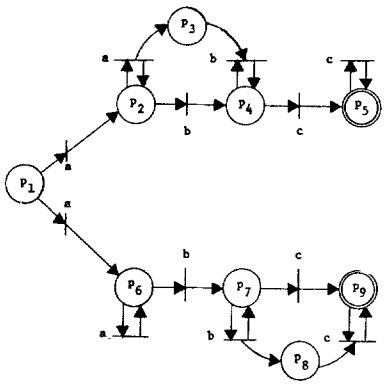
The sequences t_1t_4, $t_2t_3t_3t_4$, and t_4 are neither complete nor legal.

Associated with each sequence of transitions, $x \in T^*$, is the sequence of symbols, $y \in \Sigma^*$, defined by $y = \sigma(x)$. A sequence of symbols which corresponds to a legal and complete transition sequence is a *computation sequence*. Each computation sequence represents one (or more than one) execution of the Petri net which begins with one token in the start place and ends with one token in a final place, while all other places have zero tokens both before and after the execution (although probably not during the execution). The *computation sequence set* of a Petri net is the set of all computation sequences for that net. We denote the computation sequence set of a Petri net, C, by $L(C)$. Formally,

$$L(C) = \{y \in \Sigma \mid \exists x \in T^* \text{ such that } y = \sigma(x) \text{ and } \delta(S, x) \in F\}.$$

Many Petri nets may generate the same CSS. We define two Petri nets to be *equivalent* if their CSS are equal. The CSS is the language of the Petri net and is considered the characterizing feature of the net.

The next-state function is again extended to be defined over computation sequences as well as transition sequences by defining $\delta(q, y) = q'$ for any string $y \in \Sigma^*$ for which there exists a transition sequence, $x \in T^*$, with $y = \sigma(x)$ and $\delta(q, x) = q'$. Note that with this definition δ may no longer be single-valued, but may yield a set of states. If δ is not single-valued, then the Petri net is *nondeterministic*. We define a CSS to be nondeterministic if every Petri net which generates it is nondeterministic. A deterministic CSS is then a CSS for which there exists a deterministic Petri net which generates it. Figure 4 is a nondeterministic Petri net with a nondeterministic CSS.



$L(G) = \{ a^i b^j c^k \mid i = j \text{ or } j = k,\ i, j, k \geq 1 \}$

FIG. 4. An inherently nondeterministic Petri net.

(The proof that no equivalent Petri net is deterministic is similar to the proof in [4] that this CSS is an inherently nondeterministic context-free language.)

3. Closure Properties of Computation Sequence Sets

Having defined the Petri net automaton and its associated language, we turn now to investigating the properties of the class of CSS languages. We begin our investigation by considering the closure properties of CSS under union, intersection, concatenation, and concurrent composition. We first define a restricted class of Petri nets whose special properties are convenient in the proofs of closure under these forms of composition.

The general definition of Petri nets in Section 2 allows the construction of "pathological" Petri nets, such as the net of Fig. 5, whose strange properties make the proofs

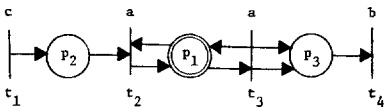


FIG. 5. A "pathological" Petri net.

which follow unnecessarily complicated. In particular, the transitions with empty input or output bags require special attention. We avoid these problems by showing that such transitions can be eliminated without changing the language of the Petri net. This is done by introducing a new place, p_r, to the net. This place is made an input and output to every transition in the net. As long as there is a token in this place, the

possible transition sequences are identical to the transition sequences of the original net; when this token is removed, all transitions are disabled. Using this approach we introduce a new start place S' and final place, p_f. New transitions are added which mimic the old transitions except that the first transition to fire places a token in p_r, and the last transition to fire removes this token. From this construction, we define a restricted class of Petri nets in *standard form* by

DEFINITION. A Petri net, $C = (P, T, \Sigma, S, F)$ is in standard form if

(1) $I(t_j) \neq \varnothing$ and $O(t_j) \neq \varnothing$ for all $t_j \in T$,

(2) $S \notin O(t_j)$ for all $t_j \in T$,

(3) there exists a place $p_f \in P$ such that

  (a) $F = \{p_f\}$ (if $\epsilon \notin L(C)$) or $F = \{p_f, S\}$ (if $\epsilon \in L(C)$),

  (b) $p_f \notin I(t_j)$ for all $t_j \in T$,

  (c) $\delta(q, t_j)$ is undefined for all $t_j \in T$ and $q \in Q$ which have a token in p_f (i.e., $q_f > 0$).

A Petri net in standard form has no transitions with empty input or output bags. It also has a start place which is an output of no transition and a special "final" place which is an input to no transition.

The execution of a Petri net in standard form starts with one token in the start place. The first transition removes this token and after this firing the start place is always empty. Eventually (if the transition sequence is complete) a token is placed in the final place. This token cannot be removed from the final place both because no transition has an input from the final place and because all transitions are disabled. The restrictive nature of the standard form Petri nets is useful when defining compositions of Petri nets. To show that standard form Petri nets are not less powerful than general Petri nets, we prove the following theorem.

THEOREM 1. *Every Petri net is equivalent to a Petri net in standard form.*

Let $C = (P, T, \Sigma, S, F)$ be a Petri net. Define $C' = (P', T', \Sigma, S', F')$ by

$$\begin{aligned} P' &= P \cup \{S', p_r, p_f\}, && \text{where } \{S', p_r, p_f\} \cap P = \varnothing, \\ F' &= \{S', p_f\} && \text{if } S \in F, \\ &= \{p_f\} && \text{if } S \notin F. \end{aligned}$$

We define four kinds of transitions in the set T'. First, for all $t_j \in T$, we include a transition $t_j' = (\sigma(t_j), I(t_j) + \{p_r\}, O(t_j) + \{p_r\})$ in T'. To start the net we consider

that two kinds transitions in T could fire first; those with $I(t_j) = \{S\}$ and those with $I(t_j) = \varnothing$. For each of these we define t_j'' by

$$\begin{aligned} t_j'' &= (\sigma(t_j), \{S'\}, O(t_j) + \{S, p_r\}) \qquad \text{if } I(t_j) = \varnothing, \\ &= (\sigma(t_j), \{S'\}, O(t_j) + \{p_r\}) \qquad \text{if } I(t_j) = \{S\}. \end{aligned}$$

Similarly the last transition to fire could be either a transition with $O(t_j) = \varnothing$ or $O(t_j) = \{p_k\}$ such that $p_k \in F$. For each of these we define t_j''' by

$$t_j''' = (\sigma(t_j), \{p_r\} + I(t_j), \{p_f\}).$$

These transitions define a legal and complete transition sequence

$$t_{j_1}'' t_{j_2}' t_{j_3}' \cdots t_{j_{l-1}}' t_{j_l}''' \qquad (l > 2)$$

in C' for every legal and complete sequence $t_{j_1} t_{j_2} \cdots t_{j_l}$ in C. In addition, we must consider sequences of length 1. For any $\sigma \in \Sigma$ for which $\delta(S, \sigma) \in F$, we add to T' a transition $(\sigma, \{S'\}, \{p_f\})$. This completes the specification of T'. From the construction, the languages of C and C' are equal, and hence, the two Petri nets are equivalent. C' is in standard form. Figure 6 illustrates the construction on the Petri net of Fig. 5. We now proceed to investigate the closure properties of CSS.

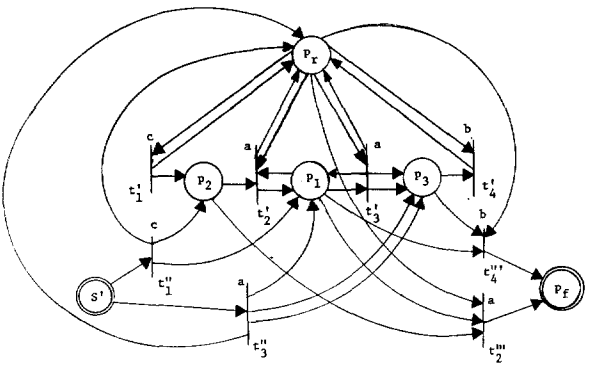
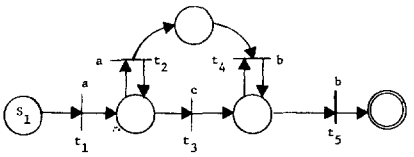
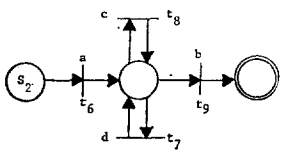


FIG. 6. A standard form Petri net equivalent to the Petri net of Fig. 5.

We consider two CSS L_1 and L_2 and two Petri nets in standard form, $C_1 = (P_1, T_1, \Sigma, S_1, F_1)$ and $C_2 = (P_2, T_2, \Sigma, S_2, F_2)$ with $L_1 = L(C_1)$ and $L_2 = L(C_2)$. We construct a new Petri net, $C' = (P', T', \Sigma, S', F')$ whose language, $L' = L(C')$, is the desired composition of L_1 and L_2. Figure 7 gives example Petri nets for C_1 and C_2 which we use in our discussions to illustrate the construction of C'.



(a) $L(C_2) = \{a^n c b^n \mid n \geq 1\}$



(b) $L(C_2) = \{a(c+d)^* b\}$

FIG. 7. Illustration Petri nets.

3.1. *Concatenation*

The concatenation of two languages can be formally expressed as

$$L_1 L_2 = \{x_1 x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

THEOREM 2. *If L_1 and L_2 are* CSS, *then the concatenation of L_1 and L_2 is* CSS.

We define a Petri net, $C' = (P', T', \Sigma, S', F')$, where

$$\begin{aligned}
P' &= P_1 \cup P_2 , \\
T' &= T_1 \cup T_2 \cup \{(\sigma_j , \{p_f\}, O_j) \mid (\sigma_j , \{S_2\}, O_j) \in T_2 , p_f \in F_1\}, \\
S' &= S_1 , \\
F' &= F_2 \qquad \text{if} \quad S_2 \notin F_2 , \\
&= F_1 \cup F_2 \qquad \text{otherwise.}
\end{aligned}$$

With this definition we have overlapped the final places of C_1 with the start place of C_2. The transition which signals the termination of C_1 by placing a token in an element of F_1 acts to initiate C_2 by placing a token in a place equivalent to S_2. Since both nets are in standard form, all transitions of the C_1 subnet are disabled when the token is placed in a final place of F_1, and all transitions of the C_2 subnet are disabled until a token is placed in one of these places. Any "extra" tokens produced by an execution of the C_1 subnet remain in that net after the token is placed in an element

of F_1, so that C' cannot reach a final state unless both C_1 and C_2 have reached final substates. Thus, if a sentence is generated by C', it must be composed of a sentence which was generated by C_1 followed by a sentence generated by C_2, and is in the concatenation of L_1 and L_2. Similarly, any computation sequence in the concatenation has a path from S_1 to an element of F_2 in C', and is an element of L'. This shows that CSS are closed under concatenation. Figure 8 illustrates this construction.

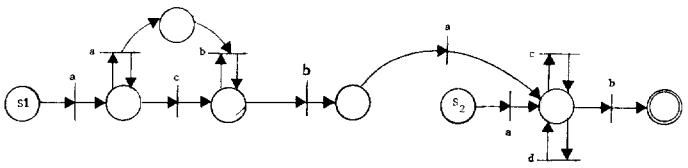


FIG. 8. A Petri net whose CSS is the concatenation of the Petri nets of Fig. 7.

3.2. *Union*

Since languages are sets of strings, a common method of composition is to take the union of two languages. This is defined as

$$L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}.$$

THEOREM 3. *If L_1 and L_2 are* CSS, *then the union of L_1 and L_2 is* CSS.

We construct C' with $L(C') = L_1 \cup L_2$. The definition of C' is

$$\begin{aligned} P' &= P_1 \cup P_2 \cup \{S'\}, \\ T' &= T_1 \cup T_2 \cup \{(\sigma_j, \{S'\}, O_j) \mid (\sigma_j, \{S_1\}, O_j) \in T_1 \text{ or } (\sigma_j, \{S_2\}, O_j) \in T_2\}, \\ F' &= F_1 \cup F_2 \cup \{S'\} \qquad \text{if } S_1 \in F_1 \text{ or } S_2 \in F_2, \\ &= F_1 \cup F_2 \qquad \text{otherwise.} \end{aligned}$$

This construction introduces one new start place and transitions which make this new start place equivalent to both S_1 and S_2. Placing the start token in S' enables a transition corresponding to every transition which would be enabled by placing a start token in S_1 or S_2. When one of these transitions fires, the output tokens are placed in a subnet defined by (P_1, T_1) or (P_2, T_2) and execution continues exactly as it would in C_1 or C_2. The null sequence is included by the definition of F'. This construction generates $L_1 \cup L_2$. Thus CSS are closed under union. The construction of C' from C_1 and C_2 is illustrated in Fig. 9 for the C_1 and C_2 of Fig. 7.

3.3. *Intersection*

As with union, the intersection composition is similar to the set theory definition of intersection and is given for CSS by

$$L_1 \cap L_2 = \{x \mid x \in L_1 \text{ and } x \in L_2\}.$$

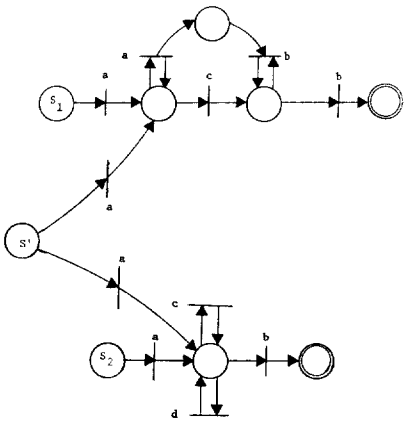


FIG. 9. A Petri net whose CSS is the union of the CSS of the Petri nets of Fig. 7.

THEOREM 4. *If* L_1 *and* L_2 *are* CSS, *then the intersection of* L_1 *and* L_2 *is* CSS.

The construction of a Petri net to generate the intersection of two CSS is rather complex. At a given point in a computation sequence if a transition fires in one Petri net, there must be a transition in the other Petri net with the same label which can fire also. When there exists more than one transition in each Petri net with the same label, we consider all possible pairs of transitions from the two nets. For each of these pairs, we create a new transition which can fire if and only if both transitions in the old nets can fire. This is done by making the input (output) bag of the new transition the bag sum of the input (output) bags of the pair of transitions from the old Petri nets. Thus if $t_j \in T_1$ and $t_k \in T_2$ are such that $\sigma(t_j) = \sigma(t_k) = \sigma_{jk}$, then we have a transition $t_{jk} = (\sigma_{jk}, I_j + I_k, O_j + O_k)$ in T'. Some of these transitions will have inputs which include the start place. If for a transition t_{jk} in T' as defined above, $I(t_{jk}) = \{S_1, S_2\}$, then we add a transition t'_{jk} with $I(t'_{jk}) = \{S'\}$, and other components equal. Similarly, for any transition t_{jk} with $O(t_{jk}) = \{p_{f_1}, p_{f_2}\}$ with $p_{f_1} \in F_1$ and $p_{f_2} \in F_2$, we add a new transition t''_{jk} which is equal to t_{jk} except that $O(t''_{jk}) = \{p_f'\}$. F' is $\{p_f', S'\}$ if $S_1 \in F_1$ and $S_2 \in F_2$ and $\{p_f'\}$ otherwise. Figure 10 illustrates this construction.

3.4. *Concurrency*

Concurrent composition allows all possible interleavings of a computation sequence from one CSS with a computation sequence from another CSS. Riddle [19] has introduced the Δ operator to represent this concurrency. The concurrency operator has also been called the "shuffle" operator [5]. It is defined for two strings by

$$ax_1 \Delta bx_2 = a(x_1 \Delta bx_2) + b(ax_1 \Delta x_2),$$
$$a \Delta \epsilon = \epsilon \Delta a = a,$$

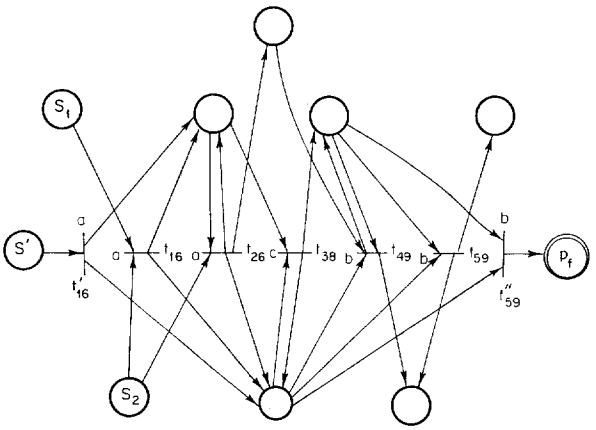


FIG. 10. A Petri net whose CSS is the intersection of the Petri nets of Fig. 7.

where $a, b \in \Sigma$, and $x_1, x_2 \in \Sigma^*$. The concurrent composition of two languages is then

$$L_1 \mathbin{\Delta} L_2 = \{x_1 \mathbin{\Delta} x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

For example, $ab \mathbin{\Delta} c = abc + acb + cab$, $(a + b) \mathbin{\Delta} c = ac + ca + bc + cb$. (The shuffle operator was defined so that it appears that strict alternation of elements of two strings is required. That is, if $x = x_1 x_2 \cdots x_k$ and $y = y_1 y_2 \cdots y_k$, then *shuffle* $(x, y) = x_1 y_1 x_2 y_2 \cdots x_k y_k$. However, x_i and y_i are allowed to be (possible null) strings, not simply elements, of the alphabet.)

It is easily shown that regular, context-sensitive and type-0 languages are closed under concurrency, while context-free languages are not. For CSS, we have

THEOREM 5. *If L_1 and L_2 are* CSS, *then the concurrent composition of L_1 and L_2 is* CSS.

The construction of a Petri net to generate the concurrent composition of L_1 and L_2 given nets to generate these CSS is basically the construction of a Petri net which places tokens in both the start places of C_1 and C_2, and then accepts the input if tokens are in any two final places (one from each net), and no other places. To start the combined Petri net we introduce a new start place, S'. The first transition which fires in the concurrent composition of two CSS will come from either C_1 or C_2. If the first transition which fires is from C_1, then we modify it to also place a token in S_2, allowing the Petri net C_2 to then start whenever it wishes. A similar strategy is used if the first transition is from C_2. Thus C' is defined by

$$P' = P_1 \cup P_2 \cup \{S', p_f'\},$$
$$T' = T_1 \cup T_2 \cup T_{SF},$$
$$F' = \{p_f'\},$$

where,

$$T_{SF} = \{(\sigma_j, \{S'\}, O_j + \{S_2\}) \mid (\sigma_j, \{S_1\}, O_j) \in T_1\}$$
$$\cup \{(\sigma_j, \{S'\}, O_j + \{S_1\}) \mid (\sigma_j, \{S_2\}, O_j) \in T_2\}$$
$$\cup \{(\sigma_j, I_j + \{p_k\}, \{p_f'\}) \mid (\sigma_j, I_j, \{p_f\}) \in T_1, p_f \in F_1, p_k \in F_2\}$$
$$\cup \{(\sigma_j, I_j + \{p_k\}, \{p_f'\}) \mid (\sigma_j, I_j, \{p_f\}) \in T_2, p_f \in F_2, p_k \in F_1\}.$$

The last two types of transitions added to T' by T_{SF} remove the tokens from final places in C_1 and C_2 and place them in a new final place when the last transition of the composition is fired. This construction is demonstrated in Fig. 11.

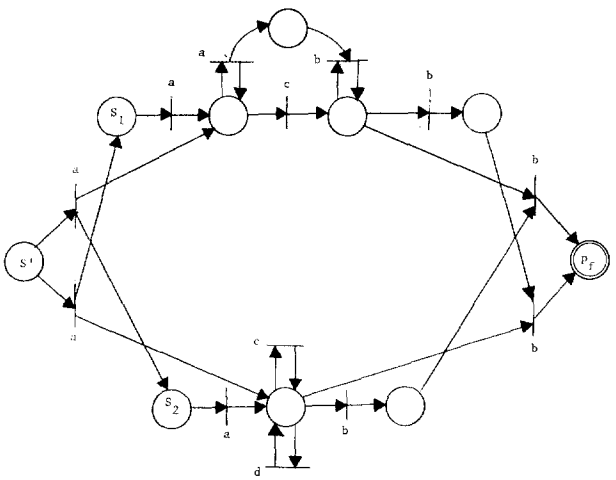


FIG. 11. A Petri net whose CSS is the concurrent composition of the Petri nets of Fig. 7.

The construction is correct only for ϵ-free CSS. However, if $L_1 = \{\epsilon\} \cup L_1^+$ with $\epsilon \notin L_1^+$, then $L_1 \Delta L_2 = L_2 \cup (L_1^+ \Delta L_2)$. Thus, since CSS are closed under union, CSS are closed under concurrent composition.

3.5. *Other Operations on* CSS

The closure properties of CSS under many other operations can be investigated, but for our purposes the above four are most relevant. It is easily shown that CSS are also closed under reversal, ϵ-free homomorphism, and ϵ-free regular substitution [17]. Hack has shown that CSS are closed under ϵ-free homomorphism, ϵ-free Finite State Transducer mappings, and inverse homomorphisms. He has also shown that CSS are *not* closed under Kleene star or general substitution [7].

It is conjectured that CSS are not closed under complement.

Hopcroft and Ullman [10] have compiled a table of closure properties of regular, context-free, context-sensitive, and type-0 languages for several closure operations. A similar study for CSS as a class of languages might shed some further light on the character of the CSS languages and indirectly, on their relationship to these other classes of languages. Knowledge of the relationship between CSS languages and these other classes of languages might be useful for establishing decidability results for CSS from the known results for these languages.

4. Comparison of CSS Languages to Other Language Classes

Thus, we turn now to investigating the relationship between CSS and the classes of regular, context-free, and context-sensitive languages.

4.1. *Regular Languages*

One of the simplest and most studied classes of formal languages is the class of regular languages. These languages are generated by regular grammars and finite state machines. They can be characterized by regular expressions. Problems of equivalence or inclusion between two regular languages are decidable and algorithms exist for their solution [10]. With such a desirable set of properties it is encouraging that we have the following theorem.

Theorem 6. *Every regular language is* CSS.

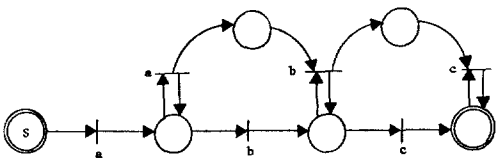
The proof of this theorem is based on the fact that every regular language is generated by some finite state machine. A finite state machine is defined as a 5-tuple, $(Q, \delta, \Sigma, S, F)$, where Q is a finite state space, δ a next-state function from $Q \times \Sigma$ into Q, Σ an alphabet, $S \in Q$ a start state, and $F \subseteq Q$ a set of final states. We can construct an equivalent Petri net as (Q, T, Σ, S, F), where the set of transitions is

$$T = \{(\sigma_i, \{q_j\}, \{q_k\}) \mid \delta(q_j, \sigma_i) = q_k\}.$$

This Petri net will generate the same language as the finite state machine. Thus, every regular language is CSS.

4.2. *Context-Free Languages*

The converse to Theorem 6 is not true. Figure 7 displays a Petri net which generates the contex-free language $\{a^n c b^n \mid n \geqslant 1\}$. Since this language is not regular, we know that not all CSS are regular. Figure 12 shows that not all CSS are context-free by exhibiting a CSS which is context-sensitive, but not context-free. Unlike the situation with regular languages, however, there also exist context-free languages which are not CSS. An example of such a language is the context-free language $\{wwv^R \mid w \in \Sigma^*\}$. This is shown in the following theorem.



$L(C) = \{a^n b^n c^n \mid n \geq 0\}$

FIG. 12. A context-sensitive, but not context-free CSS.

THEOREM 7. *There exist context-free languages which are not* CSS.

Assume there exists an n-place, m-transition Petri net which generates $\{ww^R \mid w \in \Sigma^*\}$. Let k be the number of symbols in Σ, $k > 1$. For an input string xx^R, let $l = |x|$, the length of x. Since there are k^l possible input strings x, the Petri net must have k^l *distinct* reachable states after l transitions in order to remember the complete string x. If we do not have this many states, then for some strings x_1 and x_2, we have $\delta(S, x_1) = \delta(S, x_2)$ for $x_1 \neq x_2$. Then,

$$\begin{aligned}\delta(S, x_1x_2^R) &= \delta(\delta(S, x_1), x_2^R)\\ &= \delta(\delta(S, x_2), x_2^R)\\ &= \delta(S, x_2x_2^R)\\ &\in F\end{aligned}$$

and the Petri net will incorrectly generate $x_1x_2^R$.

For each transition t_j, there exists a vector v_j such that if $\delta(q, t_j)$ is defined then $\delta(q, t_j) = q + v_j$. Thus after l inputs, a Petri net will be in a state q given by

$$q = S + \sum_{i=1}^{l} v_{j_i}$$

for a sequence of transitions $t_{j_1}, t_{j_2}, \ldots, t_{j_l}$. Another way of expressing the above sum is

$$q = S + \sum_{j=1}^{m} a_j v_j,$$

where a_j is the number of times transition t_j occurs in the sequence. We have also the constraint that

$$\sum_{j=1}^{m} a_j = l.$$

At best the vectors $v_1, v_2, \ldots, v_m$ will be linearly independent and each vector of coefficients $(a_1, a_2, \ldots, a_m)$ will represent a unique state q. Since the sum of the coefficients is l, the vector of coefficients is a partition of the integer l into m parts. Knuth [13] gives the number of partitions of an integer l into m parts as

$$\binom{l+m-1}{m-1}.$$

Now since

$$\binom{l+m-1}{m-1} = \frac{(l+m-1)\cdots(l+1)}{(m-1)!} < (l+m)^m,$$

there are strictly less than $(l+m)^m$ reachable states in Q after l inputs. For large enough l, we have then that

$$\binom{l+m-1}{m-1} < (l+m)^m < k^l.$$

It is impossible for there to be k^l distinct states in Q for each of the k^l possible input strings. Thus it is impossible for a Petri net to generate the set ww^R. Notice that this proof depends only on the number of places, transitions, and symbols. It is not affected by the deterministic or nondeterministic nature of the net.

Having shown that not all context-free languages are CSS and not all CSS are context-free, the question arises, What is the class of languages which are both context-free and CSS? At present we cannot fully answer this question, but we can give an indication of some of the members of this intersection. One subset of both classes of languages is regular languages. Another subset is the set of bounded context-free languages [4].

4.3. *Bounded Context-Free Languages*

A context-free language, L, is a *bounded context-free language* over an alphabet Σ, if there exist strings $w_1, w_2, \ldots, w_m$ from Σ^* such that

$$L \subseteq w_1^* w_2^* \cdots w_m^*.$$

Ginsburg [4] has developed a detailed examination of the properties of bounded context-free languages and gives the following characterization theorem ([4, Theorem 5.4.1]).

THEOREM 8. *The family of bounded context-free languages is the smallest family of sets defined by*

(1) *If W is a finite subset of Σ^*, then W is a bounded context-free language.*

(2) *If W_1 and W_2 are bounded context-free, then $W_1 \cup W_2$ and W_1W_2 are bounded context-free.*

(3) *If W is bounded context-free, and $x, y \in \Sigma^*$, then $\{x^iWy^i \mid i > 0\}$ is bounded context-free.*

We have already shown that every regular language (and hence every finite subset of Σ^*) is CSS. We have also shown that CSS are closed under union and concatenation. Thus we have only to show that CSS are closed under the operation described in (3) above to show that bounded context-free languages are CSS.

For any case where x, y, or W is ϵ, x^iWy^i reduces to a language of the form x^*W, Wy^*, x^*, x^iy^i, or W which are CSS, for $x, y \in \Sigma^*$ and W CSS. For nonnull x and y, we define C_x and C_y by

$$x = x_1x_2 \cdots x_k, \qquad x_i \in \Sigma, \qquad\qquad y = y_1y_2 \cdots y_l, \qquad y_i \in \Sigma,$$

$$C_x = (P_x, T_x, \Sigma, S_x, F_x), \qquad\qquad C_y = (P_y, T_y, \Sigma, S_y, F_y),$$

$$P_x = \{p_{x_1}, p_{x_2}, \ldots, p_{x_{k+1}}\}, \qquad\qquad P_y = \{p_{y_1}, p_{y_2}, \ldots, p_{y_{l+1}}\},$$

$$T_x = \{(x_i, \{p_{x_i}\}, \{p_{x_{i+1}}\} \mid 1 \leqslant i \leqslant k\}, \qquad T_y = \{(y_i, \{p_{y_i}\}, \{p_{y_{i+1}}\} \mid 1 \leqslant i \leqslant l\},$$

$$S_x = p_{x_1}, \qquad\qquad S_y = p_{y_1},$$

$$F_x = \{p_{x_{k+1}}\}, \qquad\qquad F_y = \{p_{y_{l+1}}\}.$$

With these definitions, $L(C_x) = \{x\}$ and $L(C_y) = \{y\}$. Let $C_W = (P_W, T_W, \Sigma, S_W, F_W)$ be a Petri net in standard form with $L(C_W) = W$; then we define $C' = (P', T', \Sigma, S', F')$ by

$$P' = P_x \cup P_y \cup P_W \cup \{p\},$$
$$T' = T_x \cup T_y \cup T_W \cup T_{xx} \cup T_{xW} \cup T_{Wy} \cup T_{yy},$$
$$S' = S_x,$$
$$F' = F_y,$$

where

$$T_{xx} = \{(x_k, \{p_{x_k}\}, \{p, p_{x_1}\})\},$$
$$T_{xW} = \{(\sigma(t_j), \{p_{x_1}\}, O(t_j)) \mid t_j \in T_W \text{ and } I(t_j) = S_W\},$$
$$T_{Wy} = \{(\sigma(t_j), I(t_j), \{p_{y_{l+1}}\}) \mid t_j \in T_W \text{ and } O(t_j) \in F_W\},$$
$$T_{yy} = \{(y_1, \{p, p_{y_{l+1}}\}, \{p_{y_2}\})\}.$$

The place p acts as a counter of the number of times that x has been generated and assures that y will be generated the same number of times if the string is correct. The additional transitions allow the proper sequencing of the C_x, C_W, and C_y nets.

With this construction, all bounded context-free languages are shown to be CSS. Are there context-free languages which are also CSS but not bounded? Unfortinately, yes. Ginsburg shows that the regular expression $(a + b)^*$ is not bounded context-free. Since this language is both context-free and CSS, we see that bounded context-free languages are a proper subset of the family of languages which are both CSS and context-free. $(a + b)^*ca^nb^n$ is both context-free and CSS but neither regular nor bounded.

4.4. *Context-Sensitive Languages*

We turn now to context-sensitive languages. From the example in Fig. 12 we know that some CSS are context-sensitive; below we prove that all CSS are context-sensitive. Since we know that all context-free languages are also context-sensitive and there exist context-free languages which are not CSS, there exist context-sensitive languages which are not CSS. Thus the inclusion is proper.

THEOREM 9. *All* CSS *are context-sensitive.*

There are two ways to show that a language is context-sensitive: Construct a context-sensitive grammar which generates it, or specify a nondeterministic linear bounded automaton which recognizes it. We use the latter technique for the proof given here. A proof using a context-sensitive grammar is given in [17].

A linear bounded automaton is similar to a Turing machine. It has a finite state control, a read/write head, and a (two-way infinite) tape. The limiting feature which distinguishes it from a Turing machine is that the amount of tape which can be used by the linear bounded automaton to recognize a given input string is bounded by a linear function of the length of the input string. In this sense it is similar to the pushdown automaton used to recognize context-free languages (since the maximum length of the stack is bounded by a linear function of the input string length) except that the linear bounded automaton has random access (in the same sense as a Turing machine) to its memory, while the pushdown automaton has access to only one end of its memory.

To recognize a CSS with a linear bounded automaton, we simulate the Petri net by remembering, after each input, the number of tokens in each place. How fast can the number of tokens in a Petri net grow, as a function of the length of the input? After the transition sequence $t_{j_1}, t_{j_2}, \ldots, t_{j_l}$ we have seen that the Petri net is in a state defined by

$$q = \delta(S, t_{j_1}, \ldots, t_{j_l}) = S + \sum_{i=1}^{l} v_{j_i},$$

where v_j is the vector describing the change in state caused by firing transition t_j. Since the v_j are fixed by the structure of the Petri net, there is a maximum vector ν which is (component-wise) greater than all v_j $(1 \leqslant j \leqslant m)$. Thus

$$q < S + l \cdot \nu.$$

If $|\nu| = \sum_{i=1}^{n} \nu_i$, then the number of tokens, η, in a Petri net after l transitions is bounded by

$$\eta < 1 + l \cdot |\nu|.$$

Thus the number of tokens, and the amount of memory needed to remember them, is bounded by a linear function of the input length. Hence CSS can be recognized by linear bounded automata, showing that CSS are context-sensitive.

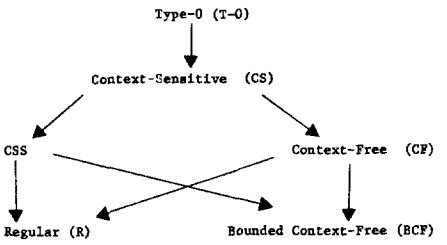


FIG. 13. Relationship of CSS to other classes of languages.

Figure 13 summarizes the relationships among the classes of languages which are regular, bounded context-free, CSS, context-free, and context-sensitive. An arc between two classes of languages indicates proper containment.

5. Decidability Problems and Conclusions

A large number of problems for CSS and Petri nets are currently unanswered. The decidability of the following list of decision problems (among others) needs resolution.

Given two CSS, are they equal? (The Equality Problem)

Given two CSS, is one a subset of the other? (The Containment Problem)

Given a CSS, is it regular, bounded context-free, or context-free?

Given a CSS, is it finite or infinite?

Given a CSS, is it empty?

The last problem above is the *emptiness problem* for CSS. This problem is central to the decidability properties of CSS languages. If the emptiness problem is undecidable, then all of the above questions are undecidable [17].

Another viewpoint on the emptiness problem for CSS can be obtained by considering the equivalence between the state space of the Petri net and vector replacement systems. Keller [12] has defined a vector replacement system as a triple (q_0, U, V),

where U and V are sets of n-vectors over the integers, with $u_j \leqslant v_j$ for $u_j \in U$ and $v_j \in V$ $(1 \leqslant j \leqslant |U| = |V|)$. A reachability set, Q, is defined by

(a) $q_0 \in Q$,

(b) if $x \in Q$ and $x + u_j \geqslant 0$, then $x + v_j \in Q$ $(u_j \in U, v_j \in V)$.

Comparing this with the definition of the state space of a Petri net (Section 2.3), we see that the emptiness problem for CSS is similar to the *reachability problem* for vector replacement systems: Given a vector replacement system with reachability set Q and an arbitrary vector x, is $x \in Q$? This reachability problem is equivalent to the reachability problem for vector addition systems [11, 14].

A short proof along the lines of Nash's proof of the equivalence of the (general) reachability to the zero reachability problem [11] shows that the emptiness problem for CSS is equivalent to the reachability problem for vector replacement and addition systems. The decidability of these questions is an open problem.

The use of concepts from formal language theory in the investigation of Petri nets is still a new field of research. Some preliminary investigations along this line have been made by other researchers. Baker [1] considered briefly the prefix languages of Petri nets defined by the set of legal (but not necessarily complete) computation sequences. This has been developed further by Hack [7], who considers the properties of four related classes of languages which can be defined for Petri nets. These languages result from considering either prefix or final-state languages either with or without null labels $(\sigma(t_j) = \epsilon)$.

Another interesting connection between formal language theory and Petri nets has been considered by Crespi-Reghizzi and Mandrioli [3]. Their work points out the relationship between Petri net languages and the matrix context-free languages. Petri net languages can also be related to the Szilard languages [20] for matrix context-free languages.

Although some of the fundamental properties of CSS have been established, many questions concerning CSS are still unanswered. We feel that CSS, and other classes of languages which can be associated with Petri nets, are an important new type of formal languages. CSS provide a useful bridge between formal language theory and research in the area of parallel computation using Petri nets, and, we believe, add significant new concepts to both existing theories.

APPENDIX: A Brief Theory of Bags

The theory of *bags* (also called multisets) has been developed by Cerf *et al.* [2]. Bags are an extension of the concept of sets. A bag, like a set, is a collection of elements from some domain. Unlike a set, however, an element may occur in a bag more than

once. A function, $\#(\cdot, \cdot)$, is defined on elements of a domain and bags over that domain which yields the number of occurrences of the element in the bag. That is,

$\#(x, \beta) = k \geqslant 0$ if there are exactly k occurrences of the element x in the bag β.

Since the theory of sets is included in the theory of bags (for the special case when the range of the $\#$ function is $\{0, 1\}$), we adopt most of the notation and many of the basic concepts of sets for our work with bags. Figure A lists some of the concepts of bags, gives the notation we use, and the formal definition in terms of the $\#$ function.

Concept	Notation	Definition
Membership	$x \in B$	$\#(x, B) > 0$
Size of bag	$\mid B \mid$	$\mid B \mid = \Sigma_x \#(x, B)$
Bag equality	$A = B$	$\forall x[\#(x, A) = \#(x, B)]$
Bag inclusion	$A \subseteq B$	$\forall x[\#(x, A) \leqslant \#(x, B)]$
Strict bag inclusion	$A \subset B$	$A \subseteq B$ and $A \neq B$
Bag union	$A \cup B$	$\forall x[\#(x, A \cup B) = \max(\#(x, A), \#(x, B))]$
Bag intersection	$A \cap B$	$\forall x[\#(x, A \cap B) = \min(\#(x, A), \#(x, B))]$
Bag sum	$A + B$	$\forall x[\#(x, A + B) = \#(x, A) + \#(x, B)]$
Bag difference	$A - B$	$\forall x[\#(x, A - B) = \#(x, A) - \#(x, A \cap B)]$
Empty bag	$\varnothing$	$\forall x[\#(x, \varnothing) = 0]$
Limited repetition over a domain D	D^n	$\forall B \in D^n, \forall x \in D[\#(x, B) \leqslant n], D^n \subseteq D^\infty$
The set of all bags over a domain D	D^∞	$\forall B \in D^\infty, \forall x \in B[x \in D]$

FIG. A. Concepts, notation, and definition of bags.

For bags over a finite domain, $D = \{d_1, d_2, \ldots, d_n\}$, a natural correspondence exists between a bag $\beta \in D^\infty$ and the n-vector $\Psi(\beta)$ over the nonnegative integers defined by

$$\Psi(\beta)_i = \#(d_i, \beta), \qquad 1 \leqslant i \leqslant n.$$

This is known as the Parikh mapping [15] of the bag.

ACKNOWLEDGMENTS

I gratefully acknowledge the help of Professor T. H. Bredt and the careful, helpful, and encouraging remarks of M. Hack in the preparation of this paper. Mr. Hack's review and comments have helped to correct some early errors in the paper.

REFERENCES

1. H. BAKER, Petri nets and languages, Computation Structures Group Memo 68, Project MAC, Massachusetts Institute of Technology, 1972.
2. V. G. CERF, E. B. FERNANDEZ, K. P. GOSTELOW, AND S. A. VOLANSKY, Formal control-flow properties of a graph model of computation, Report ENG-7178, Computer Science Department, University of California, Los Angeles, 1971.
3. S. CRESPI-REGHIZZI AND D. MANDRIOLI, Petri nets and commutative grammars, Rapporto Interno 74-5, Laboratorio di Calcolatori, Istituto di Elettrotecnica ed Elettronica del Politecnico di Milano, 1974.
4. S. GINSBURG, "The Mathematical Theory of Context-Free Languages," McGraw–Hill, New York, 1966.
5. S. GINSBURG AND S. GREIBACH, Principal AFL, *J. Comput. System Sci.* **4** (1970), 308–338.
6. M. HACK, Decision problems for Petri nets and vector addition systems, Computation Structures Group Memo 95, Project MAC, Massachusetts Institute of Technology, 1974.
7. M. HACK, Petri net languages, Computation Structures Group Memo 124, Project MAC, Massachusetts Institute of Technology, 1975.
8. A. W. HOLT AND F. COMMONER, Events and conditions, *in* "Record of the Project MAC Conference on Concurrent Systems and Parallel Computation," pp. 3–52, ACM, New York, 1970.
9. A. W. HOLT, H. SAINT, R. M. SHAPIRO, AND S. WARSHALL, Final report on the information systems project, Report RADC-TR-68-305, Rome Air Development Center, Griffiss Air Force Base, New York, 1968.
10. J. E. HOPCRAFT AND J. D. ULLMAN, "Formal Languages and Their Relation to Automata," Addison–Wesley, Reading, Mass., 1969.
11. R. M. KARP AND R. E. MILLER, Parallel program schemata, *J. Comput. System Sci.* **3** (1969), 167–195.
12. R. M. KELLER, Vector replacement systems: A formalism for modeling asynchronous systems, Technical Report 117, Department of Electrical Engineering, Princeton University, 1972.
13. D. E. KNUTH, "The Art of Computer Programming," Vol. 3: "Sorting and Searching," Addison–Wesley, Menlo Park, Calif., 1973.
14. B. O. NASH, Reachability problems in vector addition systems, *Amer. Math. Monthly* **80** (1973), 292–295.
15. R. J. PARIKH, On context-free languages, *J. Assoc. Comput. Mach.* **13** (1966), 570–581.
16. S. S. PATIL, Coordination of asynchronous events, Ph. D. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1970.
17. J. L. PETERSON, Modelling of parallel systems, Ph. D. Thesis, Department of Electrical Engineering, Stanford University, 1973.
18. C. A. PETRI, Kommunikation mit automaten, Ph. D. Thesis, University of Bonn, Germany, 1962 [German; English transl.: Supplement 1 to Technical Report RADC-TR-65-337, Vol. 1, Rome Air Development Center, Griffiss Air Force Base, New York, 1966.]
19. W. E. RIDDLE, The modeling and analysis of supervisory systems, Ph. D. Thesis, Department of Computer Science, Stanford University, 1972.
20. A. SALOMAA, "Formal Languages," Academic Press, New York, 1973.