The 14th Annual Allerton Conference on Circuit and System Theory. October, 1976. pp. 606-615

Characteristics of some augmented Petri nets.

J. Smith and T. Mudge.

Proc. of the 14th Ann. Allerton Conf. Circuit and System Theory,

Oct. 1976, pp. 606-615.

CHARACTERISTICS OF SOME AUGMENTED PETRI NETS[*]

J. E. SMITH[†] AND T. MUDGE
Coordinated Science Laboratory
University of Illinois, Urbana, Illinois 61801

ABSTRACT

Augmented Petri (APN) nets are defined. They are characterized by computation sequence sets (CSS). These are shown to be the recursively enumerable sets (RES). The relationship between APNs and other modified forms of Petri nets, called Coordination nets (CN) is shown. Several corollaries about APNs follow.

1. INTRODUCTION

Petri nets [Petr] can be used as models for the coordination and synchronization of computation processes. The decidability of several important questions concerning their behavior remains open. This paper introduces a modified form of Petri net called the augmented Petri net (APN) for which these problems are undecidable. It goes on to show that, from a behavioral viewpoint, APNs are similar to an already existing modified form by Petri net called Coordination nets (CN) [Pati]. As a by-product of these demonstrations it is suggested that CNs and Petri nets are not equivalent as was thought.

Finally, and as a consequence of their computation sequence sets (CSS) being the recursively enumerable sets (RES), the following corollaries are shown to be true for APNs:

1. The reachability problem is undecidable.
2. The boundedness problem is undecidable.
3. The coverability problem is undecidable.

2. DEFINITION OF THE APN

An APN, A, is a 9-tuple defined by:

$$A = \langle P,S,F,C,T,f,b,\Sigma,s\rangle$$

where,

$P = \{p_1,\ldots,p_k\}$ is a set of places.

$S \in P$ is the start place.

$F \in P$ is the final place.

$C = \{c_1,\ldots,c_\ell\}$ is the constraint set.

$c_i = \{p_{i_1},\ldots,p_{i_m}\}$ is a constraint class.

$T = \{t_1,\ldots,t_n\}$ is a set of transitions.

f: $P \times T \rightarrow N$ is the forward incidence function. (N is the set of non-negative integers.)

[*] This work was supported by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract DAAB-07-72-C-0259

[†] Currently with the Department of Electrical and Computer Engineering at the University of Wisconsin, Madison 53706.

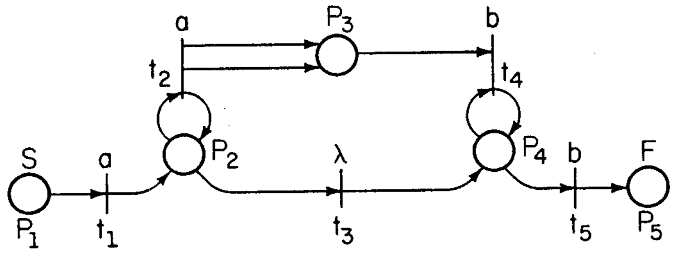
b: $P \times T \rightarrow N$ is the backward incidence function.

$\Sigma = \{s_1,\ldots,s_p\}$ is an alphabet of computations.

g: $T \rightarrow \{\Sigma \cup \lambda\}$ is the output function. (λ is the null computation.)

An APN may be represented graphically as a bi-partite graph, in which the places and transitions are represented as nodes. To distinguish them, places are represented by circles and transitions by bars. Arcs connect places to transitions and vice versa, according to the forward and backward incidence functions. If $f(p_i,t_j) = m$, then m arcs connect place p_i to transition t_j. Similarly, if $b(p_i,t_j) = n$, then n arcs connect transition t_j to place p_i. Figure 1 gives the definition of an APN together with its graphical representation.

So far just the static properties of an APN have been presented. Since an APN is an abstract machine it also has dynamic properties. These properties are exhibited during the simulation of the APN. Simulation is defined by a procedure and abstract entities called tokens, which reside at places and can be represented as dots or numbers in the place circles of the APN graph.



$P = \{P_1, P_2, P_3, P_4, P_5\}$

$S = P_1$

$F = P_5$

$C = \{P_3, P_5\}$

$T = \{t_1, t_2, t_3, t_4, t_5\}$

	t_1	t_2	t_3	t_4	t_5
P_1	1	0	0	0	0
P_2	0	1	1	0	0
P_3	0	0	0	1	0
P_4	0	0	0	1	1
P_5	0	0	0	0	0

b:	t_1	t_2	t_3	t_4	t_5
P_1	0	0	0	0	0
P_2	1	1	0	0	0
P_3	0	2	0	0	0
P_4	0	0	1	1	0
P_5	0	0	0	0	1

$\Sigma = \{a, b\}$

g	t_1	t_2	t_3	t_4	t_5
	a	a	λ	b	b

FP-5301

Figure 1. An Example APN.

Before giving the simulation procedure it is necessary to define some new terms. A transition t_i is primed if the number of tokens at each place p_j (written $|p_j|$) satisfies,

$$|p_j| \geq f(p_j,t_i)$$

A transition t_i is fireable if it is primed and the resulting new token distribution given by,

$$|p_j| \leftarrow |p_j| - f(p_j,t_i) + b(p_j,t_i) \ \forall j$$

satisfies the constraint condition viz.

$$\nexists\ c_k \in C \ (c_k = \{p_{k_1},\ldots,p_{k_s}\})$$
$$\ni |p_h| > 0 \quad \text{for } h = k_1,\ldots,k_s.$$

A transition that is fireable may fire to produce the above new token distribution or state.

The simulation procedure now follows below.

1. Initialize the APN by placing one token in the start place.
2. Compute the set of fireable transitions, U.
3. If $U = \emptyset$ then halt else:
4. Fire one transition from U.
5. Goto 2.

As an APN is simulated, transition firing sequences record the simulation. Applying the function Σ to these sequences, term by term, yields sequences of computations. Associated with each APN is a set of such sequences, any one of which results when the APN is simulated from its initial state to the state $|F| = 1$ and $p_j = 0 \ \forall j \ni p_j \neq F$. This set is called the computation sequence set [Pete] (written CSS).

Figure 1 shows an APN which has been initialized with a token in $S = p_1$. Its CSS is the sequences $a^n b^{2n-1}, n > 0$.

3. PROPERTIES OF CSSs

In this section we will show that the CSSs produced by APNs are the RES (or type 0 languages), so that APNs and Turing machines can be considered equally powerful at modelling the coordination and synchronization of computation sequences.

For this, a result of [Mati] is used. It can be stated as the following assertion:

For each $S \in$ RES, $\exists$ a diophantine polynomial (taken here to mean a polynomial with integer coefficients and integer roots) in $n + 1$ variables, $P(x,y_1,\ldots,y_n)$, such that if x is a non-negative integer encoding of any $w \in S$ then $\exists$ integers $y_1,\ldots y_n$ such that $P(x,y_1,\ldots,y_n) = 0$. Furthermore $\nexists$ integers $y_1,\ldots,y_n$ such that $P(x,y_1,\ldots,y_n) = 0$ for any x that is not an encoding of a $w \in S$.

By showing how an APN can be made to behave like such a polynomial, it can be shown that the RES form a subset of the CSSs for APNs.

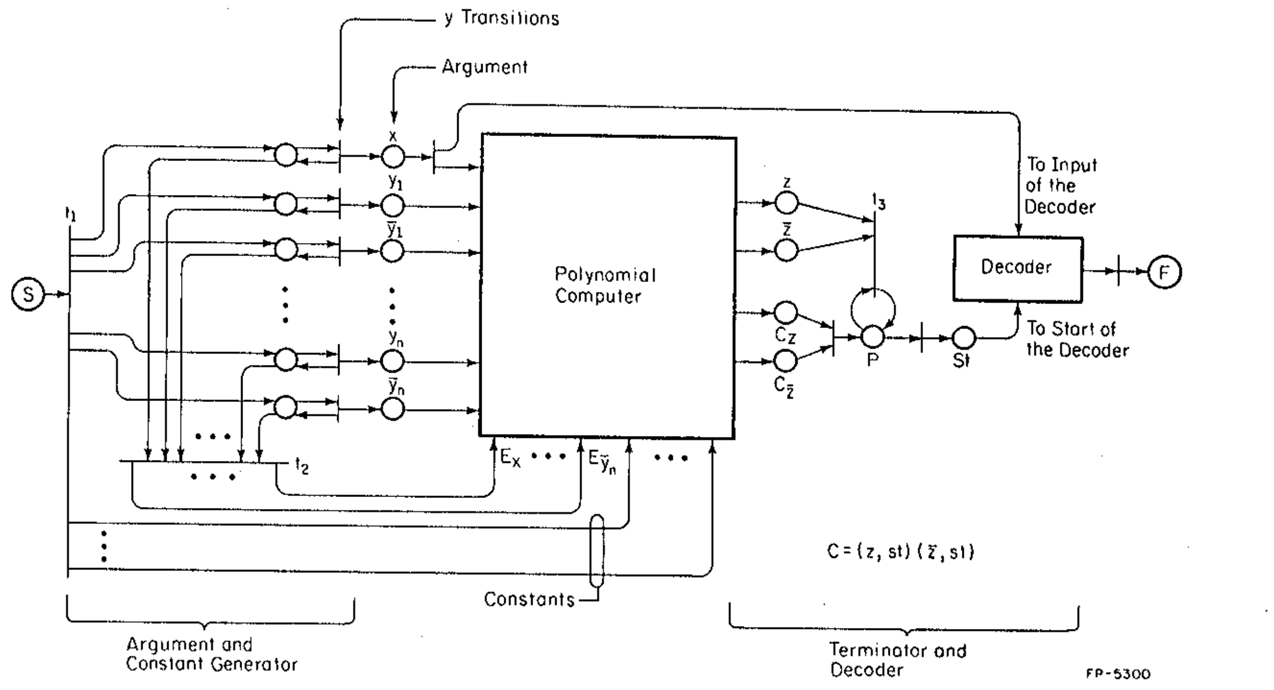


Figure 2. The Construction.

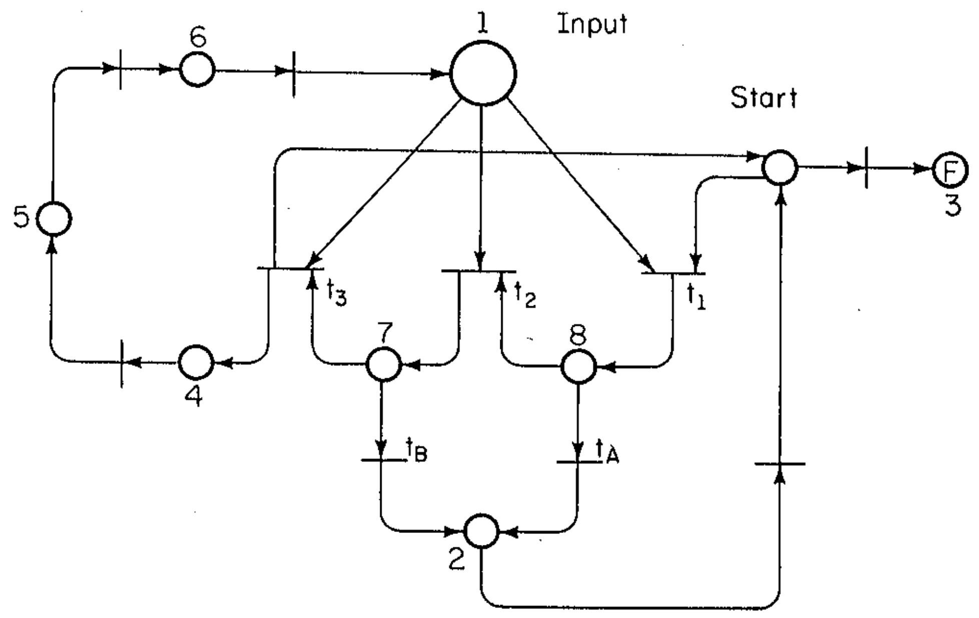
The construction is outlined in Figure 2. Initializing the start place with a token fires t_1 and activates the argument and constant generator. This causes a set of places corresponding to the integers $(x, y_1,\ldots,y_n)$ and the constants of the polynomial to be filled with tokens. So that each integer $y_i \in (y_1,\ldots,y_n)$ can range over the positive and negative integers they are encoded into two places, a positive one y_i (see Figure 2) and a negative one $\bar{y}_i$. The value of any y_i is given by the difference between the number of tokens in the positive place and the number in the negative place. Since x only ranges over the non-negative integers its value is given by the number of tokens in a single place x. The actual values of x and the y_i are determined by an arbitrary number of firings of the y transitions. This part of the machine is non-deterministic and models a search through vectors of the form $(y_1,\ldots,y_n)$ to find if there exists integer values for the y_is such that for some non-negative x, $P(x,y_1,\ldots,y_n) = 0$. Failure, for any argument $(x,y_1,\ldots,y_n)$ selected in this non-deterministic manner, results in the machine producing the λ computation sequence and not terminating properly. (Proper termination corresponds to the state $|F| = 1$ and $p_j = 0 \; \forall j \ni p_j \neq F$; see the definition of a CSS in section 2.) Success results in the machine producing a computation sequence which corresponds to an element of the RES defined by the machines polynomial. The constants produced by the argument and constant generator, for use as the integer coefficients in the polynomial computer, are encoded in a manner similar to the y_is.

Their constant value is determined by the use of multiple arcs from t_1 to their respective places.

Once the arguments and constants have been generated, transition t_2 fires, which activates the polynomial computer by placing tokens in its E (enable) inputs. The polynomial computer than computes the value of $P(x,y_1,\ldots,y_n)$, which is deposited into places z and $\overline{z}$ along with tokens in places C_z and $C_{\overline{z}}$, to signify the completion of the polynimial computation. The tokens in C_z and $C_{\overline{z}}$ initiate the terminator, which comprises transition t_3 and the places p and st, by moving to place p. Transition t_3 repeatedly fires until it can fire no more. If at this point $|z| = |\overline{z}|$ (i.e. the value of $P(x,y_1,\ldots,y_n) = 0$) the token at place p is released (this partially controlled by the constraint set) and moved to place st, the decoder can now be activated. The decoder receives an equal number of tokens as were in x (see Figure 2). These are interpreted as a non-negative encoding of a sequence of symbols from some computation alphabet Σ. In Figure 3 the details of a decoder are shown. Its construction is based on the assumption that the token count in x is a ternary encoding of sequences in $\{A,B\}^*$ (1_3 for an A and 2_3 for a B), so that, for example, a token count of 149 (12112_3) in x would represent an encoding of the sequence BAABA (reading right to left). The operation of the decoder is straightforward, the contents of place 1 are repeatedly counted down by 3, by the sequential firing of transitions t_1, t_2 and t_3. The remainder after each count down will be either one or two tokens, resulting in either t_A or t_B firing (this is partially controlled by the constraint set). The transitions t_A and t_B correspond to computations A and B, respectively, from the computation alphabet. All other transitions in the machine map to the λ computation (i.e. $g(t_A) = A$, $g(t_B) = B$ and $g(t) = \lambda\ \forall t \neq t_B, t_B$). When the sequence has been generated a token is placed in the final place F as the constraint set dictates. Of course, other encodings can be assumed and other decoders constructed in order to produce recursively enumerable sets defined on different alphabets.

Using the construction of Figure 2 any RES can in theory be generated by considering the alphabet of the set to be a computation alphabet, then selecting a non-negative encoding of that alphabet (thus defining the decoder) and then selecting the appropriate diophantine polynomial (thus defining the polynomial computer), whose integer roots coincide with those values for x that are encodings of the members of the required RES.

Elements of the polynomial computer are shown in Figure 4. There are three basic ones, an adder, a copier and a multiplier. In the adder, if x tokens are deposited in place 1, y in place 2 together with a token in E_1 and one in E_2, the adder will stop with $x+y$ tokens in place 3 and a token in place C_o. The addition will not start until a token is placed in both E_1 and E_2, and a token is not placed in C_o until places 1 and 2 are empty and the $x+y$ tokens are in place 3. The Es can be regarded as the enable places which start the basic computation and C_o as the completion place which signifies completion of the basic computation. Notice how the constraint set ensures correction operation. From the explanation of the adder's operation, the operation of the copier element should be apparent.



C = (1,2)(1,5)(1,3)(3,4)(3,5)(3,6)(5,7)(5,8)

FP-5299

Figure 3. A Decoder.

Operation of the multiplier is slightly more complicated. If x tokens are deposited in place 1 and y in place 2, then x • y are produced in place 9. The operation is synchronized by tokens in the places E_1 and E_2, and completion is signified by a token in place C_o as with adder. The element starts by moving the tokens in place 2 into place 7. These tokens are alternately moved between places 7 and 6. In transfering them from 7 to 6 transition t is fired y times causing y tokens to be deposited in place 9. This is regulated to occur exactly x times (resulting in x • y tokens in place 9) by moving one of the tokens in 1 to place 4 each time the tokens in 7 are to fire transition t y times. Place 8 is used to clear the y tokens from place 7 when place 1 is empty, prior to the completion of the multiplication.

These basic elements can be combined to form complex adders, copiers and multipliers (see Figure 5) so that signed arithmetic can be modelled, for those integers encoded into a positive and negative place. When combining basic elements, output places become the input places of subsequent elements. Similarly their corresponding C_o and E places are matched to preserve the overall sequencing of the computer.

Figure 6 shows how a polynomial computer for the polynomial xy + 3x - 2 can be built from complex adders, copiers and multipliers. The constants 3 and -2 would be generated by the argument and constant generator. In the case of constant 3, the firing of transition t_1 of the argument and constant generator (see Figure 2) would place 3 tokens into one of the positive input places of the complex multiplier shown at the top of

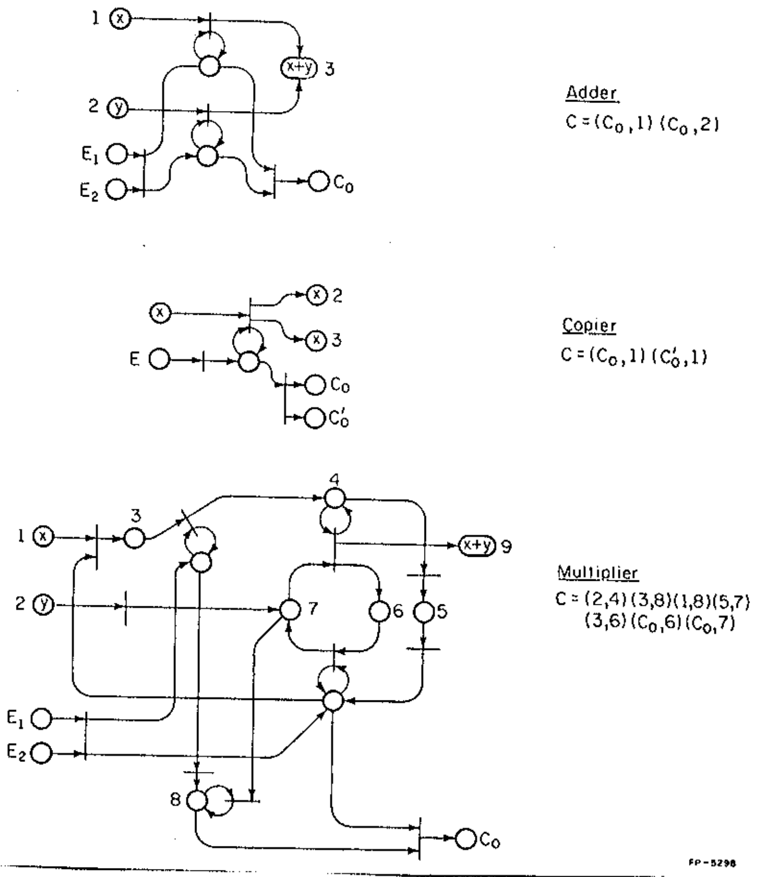


Figure 4. The Basic Elements.

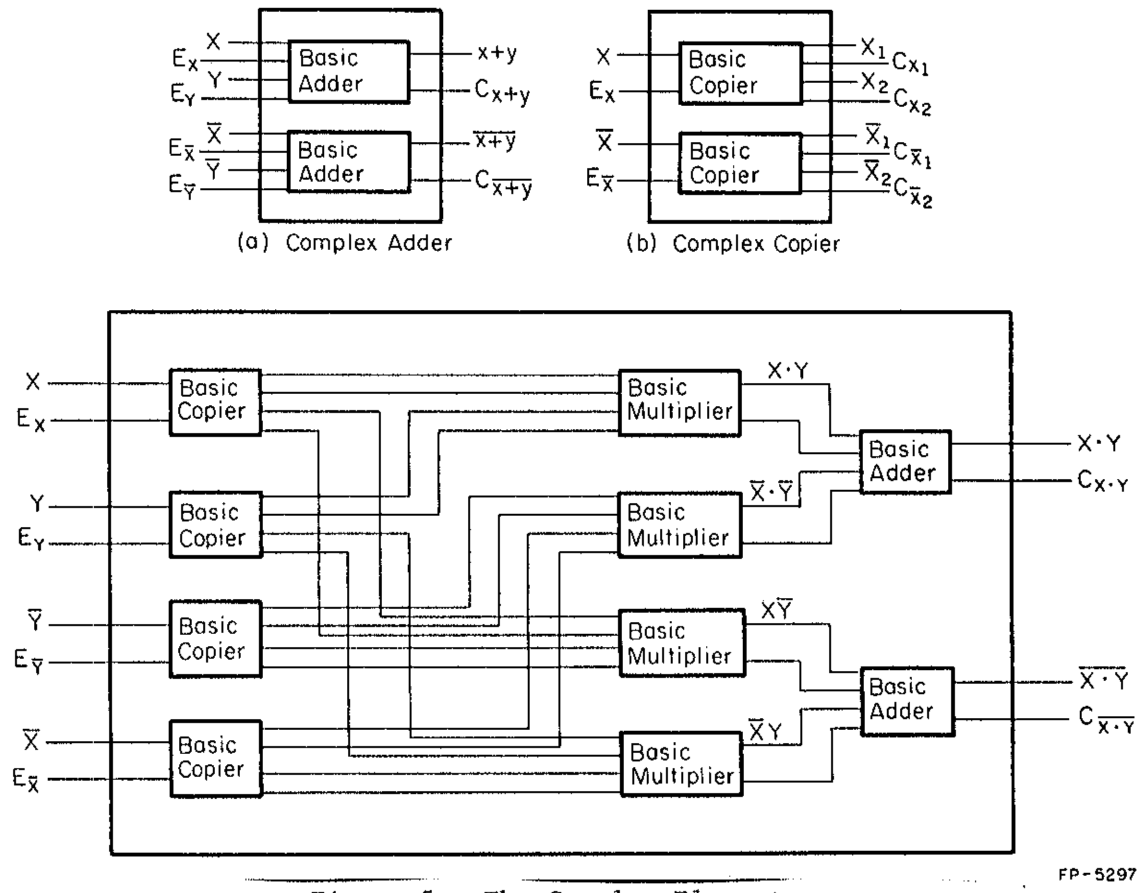


Figure 5. The Complex Elements.

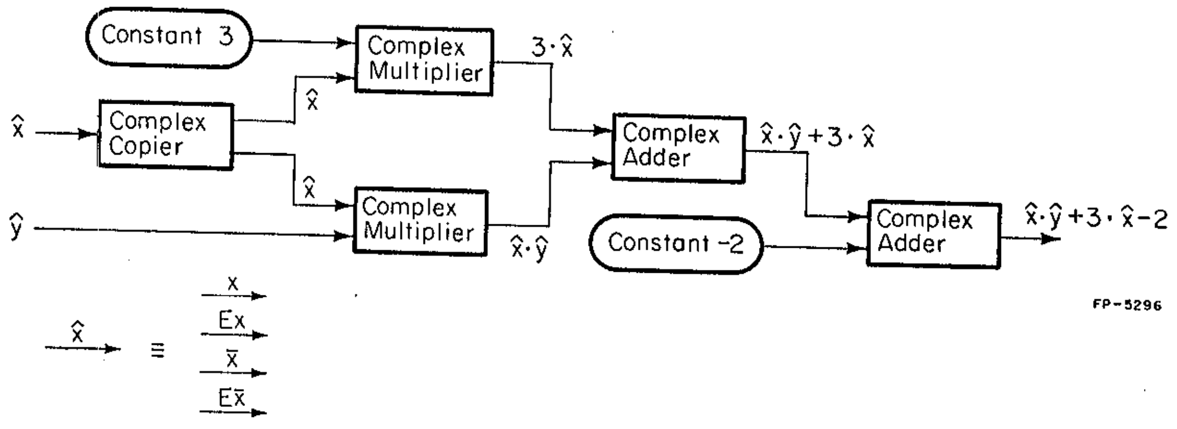


Figure 6. Example Polynomial Computer.

Figure 6. Calling this place p+ implies $b(p+,t_1) = 3$. Similarly for constant -2, except that one of the negative input places of the complex adder is used to introduce the constant.

Lemma 1: From the definition of a CSS together with the assertion at the beginning of this section and the construction method that follows it, it follows that;

$$C(CSS) \supseteq RES$$

where C(CSS) is the class of CSSs produced by all APNs.

Lemma 2: From the definition of an APN it should be clear that the class of all APNs can be enumerated. Hence the class of CSSs produced by all APNs can be enumerated, i.e.

$$RES \supseteq C(CSS)$$

Theorem: C(CSS) = RES follows from lemmas 1 and 2.

4. ON THE RELATIONSHIP BETWEEN APNs AND CNs

In the definition of APNs, we have allowed the use of multiple arcs to connect a particular transition and a particular place, i.e. the ranges of f and b are N. Nevertheless, for any APN with multiple arcs, it is possible to construct an APN with only a single arc connecting any transition and any place and which has the same CSS as the original APN. We will call such APNs restricted APNs. We say the two APNs are behaviorally equivalent since they have identical CSSs.

We now give a method for constructing a restricted APN which is behaviorally equivalent to a given APN. The method presented here is analogous to the one given by [Hack] for constructing a restricted Petri net which is equivalent to a generalized Petri net.

1. For each place, p_i, let k be the maximum number of arcs that go to or come from p_i.

2. Replace p_i with k new places, $p_{i1}, p_{i2}, \ldots p_{ik}$. Let this set of places be P_i.

3. Using single arcs and λ transitions, connect the p_{ij} into a ring.

4. Distribute the arcs going into and out of p_i in the original net over the new places p_{ij} such that no transition has more than one arc connected to any single place.

5. If the constraint set of the original APN contains the following constraint class $\{p_a, \ldots, p_z\}$ then the constraint set of the restricted APN contains the following classes $P_a X \ldots X P_z$.

Proving the above requires only a slight modification of that in [Hack]. It is easily seen from the above construction that for every restricted APN there is a behaviorally equivalent APN and vice versa.

The only difference between restricted APNs and CNs defined by Patil [Pati] is that a restricted APN has a special start place and special finish place. Peterson [Pete] effectively shows that the CSSs generated by Petri nets with start places and finish places are a subset of type 1 languages. This and the fact that the CSSs of restricted APNs must be the set of type 0 languages strongly suggest that the modelling power of CNs is stronger than that of Petri nets.

5. COROLLARIES

There are several corollaries of the Theorem in section 3 concerning the decidability of certain properties of an APN. The first such problem is reachability; that is, given an APN and a state, will the APN ever reach the given state? A second problem is boundedness; that is, for any particular place p, is there a finite number n such that the APN can never reach a state such that p has greater than n tokens? A final problem is coverability; that is, for a given APN and a given state, will the APN ever reach a state which has at least as many tokens in each place as the given state has?

Corollary 1: The reachability problem is undecidable.

Proof: The final state of an APN is only reached if the CSS of the APN is not emply. To determine emptiness of the CSS is equivalent to determining emptiness of a type 0 language. This problem is known to be undecidable.

Corollary 2: The boundedness problem is undecidable.

Proof: To the final place, F, append the places and transitions shown in Figure 7. The place F' becomes the new final place. We see that the place B becomes unbounded if F receives a token. Since it is undecidable whether F receives a token (see Corollary 1) it must be undecidable whether B is unbounded.

Corallary 3: The coverability problem is undecidable.

Proof: It is undecidable whether the final state is coverable.

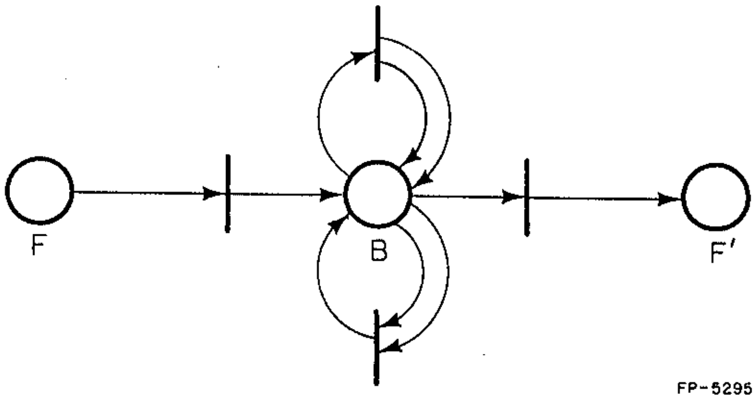


Figure 7. Corollary 2.

6. CONCLUSION

Although the behavior of APNs characterized by their CSSs, allows a greater modelling power (they can model any coordination of events which can be algorithmically described), the undecidability of the corollaries of section 5 rules out any universal techniques for answering some important questions concerning the computation processes that they might model. Since the validity of these corollaries has yet to be established for Petri nets, the same negative conclusions may not apply.

Since submitting this paper the authors became aware of similar prior work by Agerwala [Ager]. This proved the result of section 3 using a similar Petri net type machine called an Extended Petri Net. The result was achieved by showing that any Turing machine could be directly constructed as an Extended Petri net. This was also the original proof, used in this paper. However, so that this paper was not wholly replicative section 3 was redrafted with a different proof based on the results in [Mati].

7. REFERENCES

Ager Agerwala, T., "Towards a Theory for the Analysis and Synthesis of Systems Exhibiting Concurrency," Ph.D. Thesis, The Johns Hopkins Univ., 1975.

Hack Hack, M., "Decision Problems for Petri Nets and Vector Addition Systems," Technical Memo 59, Project MAC, MIT, Mar. 1975.

Mati Matijesavic, J. V., "Enumerable Sets are Diophantine," Soviet Math. Dokl., Vol. 11, No. 2 (1970), pp. 354-357.

Pati Patil, S. S., "Coordination of Asynchronous Events," Ph.D. Thesis, Dept. E.E., MIT, Jun. 1970, [MAC-TR-72].

Pete Peterson, J. L., "Modelling of Parallel Systems," Ph.D. Thesis, Dept. E.E., Stanford Univ., Dec. 1973.

Petr Petri, C. A., "Kommunikation Mit Automaten," Ph.D. Thesis, Univ. of Bonn, Germany, (1962).