Report Number: WUCS-87-26

1987-09-01

Transaction Network: A Parallel Computation Model based on Consume/Produce Paradigm

Takayuki Dan Kimura

This report introduces a new parallel computation model that is suitable for pursuit of large scale concurrency. Our goal is to develop a semantically clean paradigm for distributed computation with fine-grained parallelism. Our approach is to demote the notion of process as the key concept in organizing large scale parallel computation. We promote, instead, the notion of transaction, an anonymous atomic action void of internal state, as the basic element of computation. We propose to organize a computation as a network, called a transaction net, of databases connected by transactions. A transaction, when it is fired, consumes data objects...
Read complete abstract on page 2.

Transaction Network: A Parallel Computation Model based on Consume/Produce Paradigm

Takayuki Dan Kimura

Complete Abstract:

This report introduces a new parallel computation model that is suitable for pursuit of large scale concurrency. Our goal is to develop a semantically clean paradigm for distributed computation with fine-grained parallelism. Our approach is to demote the notion of process as the key concept in organizing large scale parallel computation. We promote, instead, the notion of transaction, an anonymous atomic action void of internal state, as the basic element of computation. We propose to organize a computation as a network, called a transaction net, of databases connected by transactions. A transaction, when it is fired, consumes data objects from source databases and produces data objects in target databases as an atomic action. A transaction net is akin to a Petrich net, where the token, the place, and the transition corresponds to the data, the database and the transaction, respectively. The state of computation is represented by the data state without the control state. An informal definition of the model is given, and solutions for well known programming problems such as sorting, transitive closure, Hamiltonean circuit, shortest path, and the eight queen's problem.

TRANSACTION NETWORK
A Parallel Computation Model based on Consume/Produce Paradigm

Takayuki Dan Kimura

WUCS-87-26

September 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

This report introduces a new parallel computation model that is suitable for pursuit of large scale concurrency. Our goal is to develop a semantically clean paradigm for distributed computation with fine-grained parallelism. Our approach is to demote the notion of process as the key concept in organizing large scale parallel computation. We promote, instead, the notion of transaction, an anonymous atomic action void of internal state, as the basic element of computation. We propose to organize a computation as a network, called a *transaction net*, of databases connected by transactions. A transaction, when it is fired, consumes data objects from source databases and produces data objects in target databases as an atomic action. A transaction net is akin to a Petri net, where the token, the place, and the transition corresponds to the data, the database and the transaction, respectively. The state of computation is represented by the data state without the control state.

An informal definition of the model is given, and solutions for well known programming problems such as sorting, transitive closure, Hamiltonean circuit, shortest path, and the eight queen's problem.

1. Introduction

In the past, a parallel computation was usually modelled by a society of processes communicating with each other, either through shared memory protected by a monitor as in Concurrent Pascal, through message passing as in the Actor model and CSP, or through remote procedure calls as in Ada. We call the underlying paradigm for these models the send/receive (SR) paradigm, because the primary communication primitives are the send/receive operations. We will propose a parallel computation model based on a different set of communication primitives; the consume/produce operations. Thus, we call the underlying paradigm for our model the consume/produce (CP) paradigm.

A process can be arbitrarily complex in terms of local data structures and control sequencing. A process's internal state consists of its control state and its data state. The state of computation is represented by the cartesian product of the individual process states. The systolic model and the dataflow model also share the same characteristic of process dominance over data.

There are at least two difficulties in scaling up such process-oriented models. The most significant one is the complexity of algorithm analysis due to the coupling between the control state and the data state. For example, the deadlock detection in a CSP program requires information about which process is in what state waiting for what data. Similarly, in systolic algorithm design, data synchronization requires the knowledge about which cell in what state carries what data and is ready to transfer them to which neighbor. Keeping track of both control state transition and data state transformation simultaneously becomes a formidable task when the number of processes increases. Our proposed solution to this problem is to make a process void of internal state and to reduce algorithm analysis to data state analysis.

The second difficulty comes from the management of process identifiers. Since the process name is required for communication through a send/receive protocol even when process names do not contribute to the computation, the programmer must manage a large name space for a large scale concurrent system. This is similar to the label management problem in assembly language programming. With structured programming constructs, programmers seldom need to use a label, while in assembly programming he is forced to use labels. Our proposed solution is to make a transaction anonymous and to replace direct

send/receive communication with indirect consume/produce interaction which requires no process identification.

The history of consume/produce paradigm of computation goes back to Post and Markov, even though parallel computation was not an issue for them. It was inherited by Production Systems [10] in AI applications and is adopted by AI programming systems such as OPS83 [3] and KEE [2] as a mechanism for forward chaining. The bridge between the consume/produce paradigm and parallel computation was proposed by the closure statements of associons [7] and generative communication in the Linda project [4]. However, in all previous known efforts, processes are not necessarily atomic and the database, called the tuple-space or working memory, is the global buffer shared by all processes. In contrast, our transaction is atomic and the database is modularized and distributed to facilitate larger degree of concurrency.

The Petri Net was originally proposed by Petri as a communication behavior model [9] and later was reintroduced as a concurrent system model by Holt and Commoner [5]. Since then, several attempts have been made to use Petri nets for modelling parallel program behavior. For example, the Colored Petri Net of Zervos [12] defines a colored token to represent an activation of reentrant code (e.g., recursion). However, no known efforts have been made to use a Petri net as a directly executable program statement. There are two major incentives for us to define a programming paradigm based on the Petri Net: a simple and clean formal semantics and a two- dimensional syntax suitable for visualization.

Visual programming is a new concept in software engineering that takes advantage of high resolution graphics capabilities for better software environments. While we have demonstrated with the Show and Tell visual programming language design [6] that no textual expressions are needed for programming, our experience suggests that a combination of textual and iconic representations of language constructs would yield better visual languages. A transaction net is a combination of a two dimensional graphic structure and textual expressions.

In the next section we will define the transaction net informally with simple examples. In Section 3 we will construct transaction nets for various programming problems, including sorting, the problems of finding the transitive closure, directed Hamiltonean circuit, shortest paths in a finite digraph, and the eight queen's problem. In the concluding section we will discuss research problems associated with the transaction net.

2. Transaction Net

We will define the transaction net here informally by using simple examples to present the basic concepts. A formal definition will be given elsewhere. Though a transaction net can be defined on any data type, in this paper we will use the integer data type as the basic data objects.

A *transaction net* is a bipartite directed graph of databases (circles) and transactions (boxes) connected by a set of arcs as schematically shown in Figure 1.

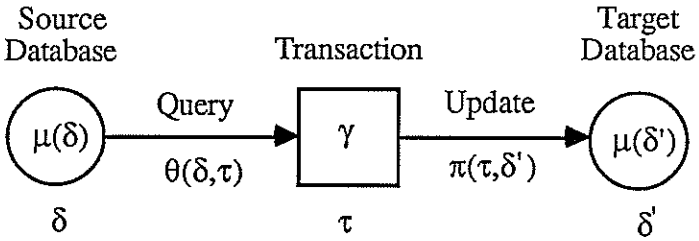

Figure 1: Scheme of Transaction Net

The marking $\mu(\delta)$ of databases δ is a set of *data expressions* each of which consists of constants and parentheses in the form of well balanced parenthesis expression representing a structured data object (a list of integers, for example). Every arc from database δ to a transaction τ is labelled by a *query expression* $\theta(\delta,\tau)$ consisting of variables and parentheses in the form of well balanced parenthesis expression. The query expression θ specifies the patterns (forms) of data objects to be consumed by τ from δ when τ fires. Similarly every arc from a transaction τ to a database δ' is labelled by an *update expression* $\pi(\tau,\delta')$ in the same form as query expressions with variables and parentheses. The update expression π specifies the data objects to be produced by τ in δ' when τ fires. The transaction box τ may be empty or may contain a *constraint expression* $\gamma(\tau)$ which is a predicate consisting of variables, constants, data operations, and logical operations $\{\wedge, \vee, \neg\}$. The constraint expression γ specifies a condition to be satisfied by all the data objects consumed and produced by a firing of the transaction. The transaction τ is *enabled to fire* when the following condition is satisfied:

There exists an assignment σ: Variables $\rightarrow$ Constants, which is a mapping from the set of variables to the set of data expressions, such that for any database δ connected to τ,

(1) $\sigma(\theta(\delta,\tau)) \subseteq \mu(\delta)$,

(2) $(\mu(\delta) - \sigma(\theta(\delta,\tau))) \cap \sigma(\pi(\tau,\delta)) = \phi$ (the empty set), and

(3) $\sigma(\gamma(\tau))$ is true,

where $\sigma(\alpha)$ denotes the result of substitution for every occurrence of variables in α by the corresponding constant under σ. For example if $\theta(\delta,\tau) \equiv \{ (x)(y), (x)((y)(x)) \}$, $\sigma(x) \equiv 2$ and $\sigma(x) \equiv 3$, then $\sigma(\theta(\delta,\tau)) \equiv \{ (2)(3), (2)((3)(2)) \}$. Thus, $\sigma(\theta(\delta,\tau))$ and $\sigma(\pi(\tau,\delta'))$ represents in general a set of data objects.

The condition (1) states that the transaction, when it fires with a chosen assignment, should be able to consume all the data objects specified by the query expression under the assignment. The condition (2) states that the data objects to be produced by the transaction should not exist in the database after the data objects to be consumed by the transaction are removed from the database. It follows that the firing condition for a transaction depends upon the markings of both target databases and source databases. This contrasts with the firing rule of the Petri net which states that a transition may fire if all the input places has at least one token, regardless of the contents of the output places. The condition (3) states that the choice of assignment, namely the choice of data objects for productions and consumptions to and from different databases connected to the transaction, must satisfy the condition associated with the transaction.

Among the set of enabled transactions, one transaction, a nondeterministically chosen one, may fire at a time. When the transaction τ *fires*, there exists at least one assignment satisfying the above three conditions, and, with a nondeterministically chosen assignment σ, the marking μ of databases changes to μ' as follows:

For any database δ, $\mu'(\delta) = \mu(\delta) - \sigma(\theta(\delta,\tau)) + \sigma(\pi(\tau,\delta))$,

where + denotes the set union operation and - denotes the set difference operation.

If there is no arc from δ to τ, then $\sigma(\theta(\delta,\tau)) \equiv \phi$ by definition, and similarly if there is no arc from τ to δ, then $\sigma(\pi(\tau,\delta)) \equiv \phi$.

It is important to note that the above definition of transaction net does not provide a net with the capability of detecting the absence of a particular data object at a particular database. By the same token, there is no direct way of testing whether a particular database is empty. In other words, it is impossible to construct a transaction net in which a particular transaction

fires when and only when the database becomes empty. This is parallel with the Petri net's incapability of counting without an introduction of the inhibitor arc.

It is also important to note that, as in the Petri net, the restriction to one firing at a time does not limit the model's capability of specifying concurrent activities in any way. We will discuss this issue further in the next section.

In order to illustrate the syntax and other notations we will use in the next section, we present in Figure 2 a sequence of transaction nets all computing the factorial function. The purpose of this example is not to demonstrate the model's capability for concurrency specification, but to explain different notations and the firing rule defined above.

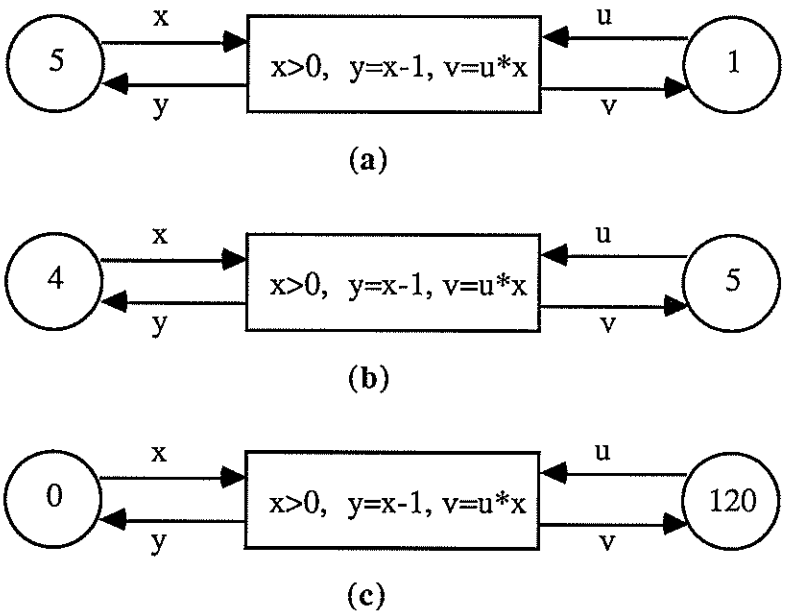

Figure 2: Factorial Functions (a) - (c)

The net (a), which has one transaction and two databases, computes the factorial of 5. The transaction is enabled by the following assignment σ: $\sigma(x) = 5$, $\sigma(y) = 4$, $\sigma(u) = 1$, $\sigma(v) = 5$. The firing with this σ will change the marking of (a) to that of net (b). By firing the transaction five times, the net reaches the quiescent marking of net (c).

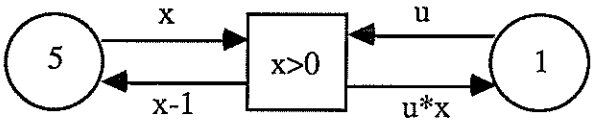

Figure 2: (d)

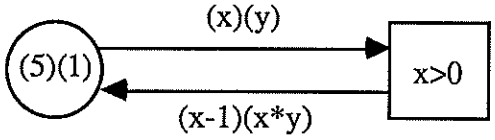

Figure 2: (e)

The net (d) is an abbreviation of net (a). In this abbreviated syntax we allow data operations in update expressions so that constraint expressions can be simplified by reducing the number of variables. The net (d) can be transformed into an equivalent net (e) using the structured data objects. Note that by a similar method any transaction net with n>0 databases can be transformed into an equivalent net with one database containing a set of n-tuples.

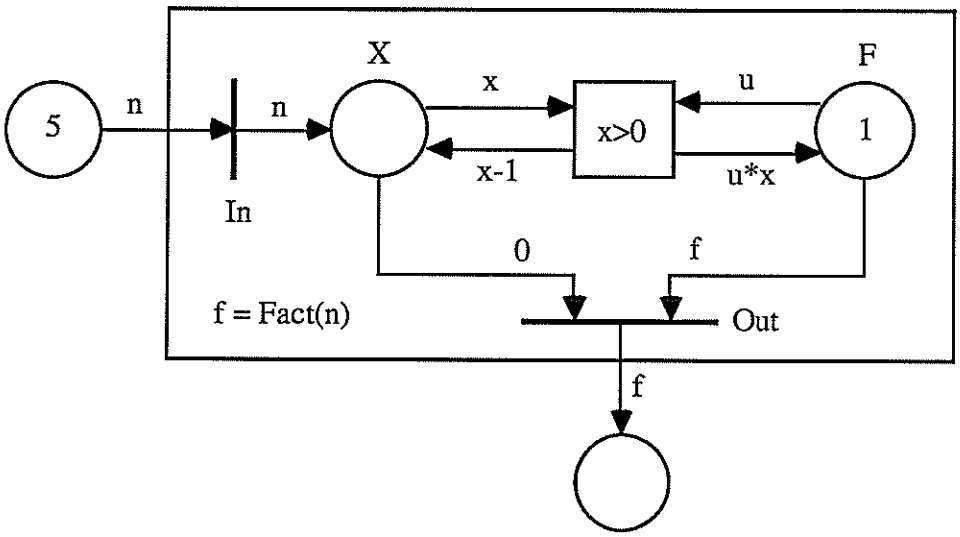

Figure 2: (f)

The net (f) contains a subnet that represents the factorial function. In this syntax we allow a constant to appear in query expressions and a straight line to represent a transaction as a transition in the Petri net. The transaction Out can fire only when the computation of 5! is completed. Note that firings of the main transaction preserves the following invariant on the markings: $\mu(X)! * \mu(F) = n!$.

3. Concurrent Computations

In this section we will present some examples of concurrent transaction nets. The factorial examples given in the previous section represent sequential computations in the sense that at all times there exists only one transaction enabled in a net and that at all times there exists at most one data object in each database. We define two enabled transactions to be *concurrent* if they are not in conflict and the order of firings of the two is insignificant. A transaction net is *concurrent* if there exists a reachable marking under which two enabled transactions are concurrent. For example, consider the nets in Figure 3 below.

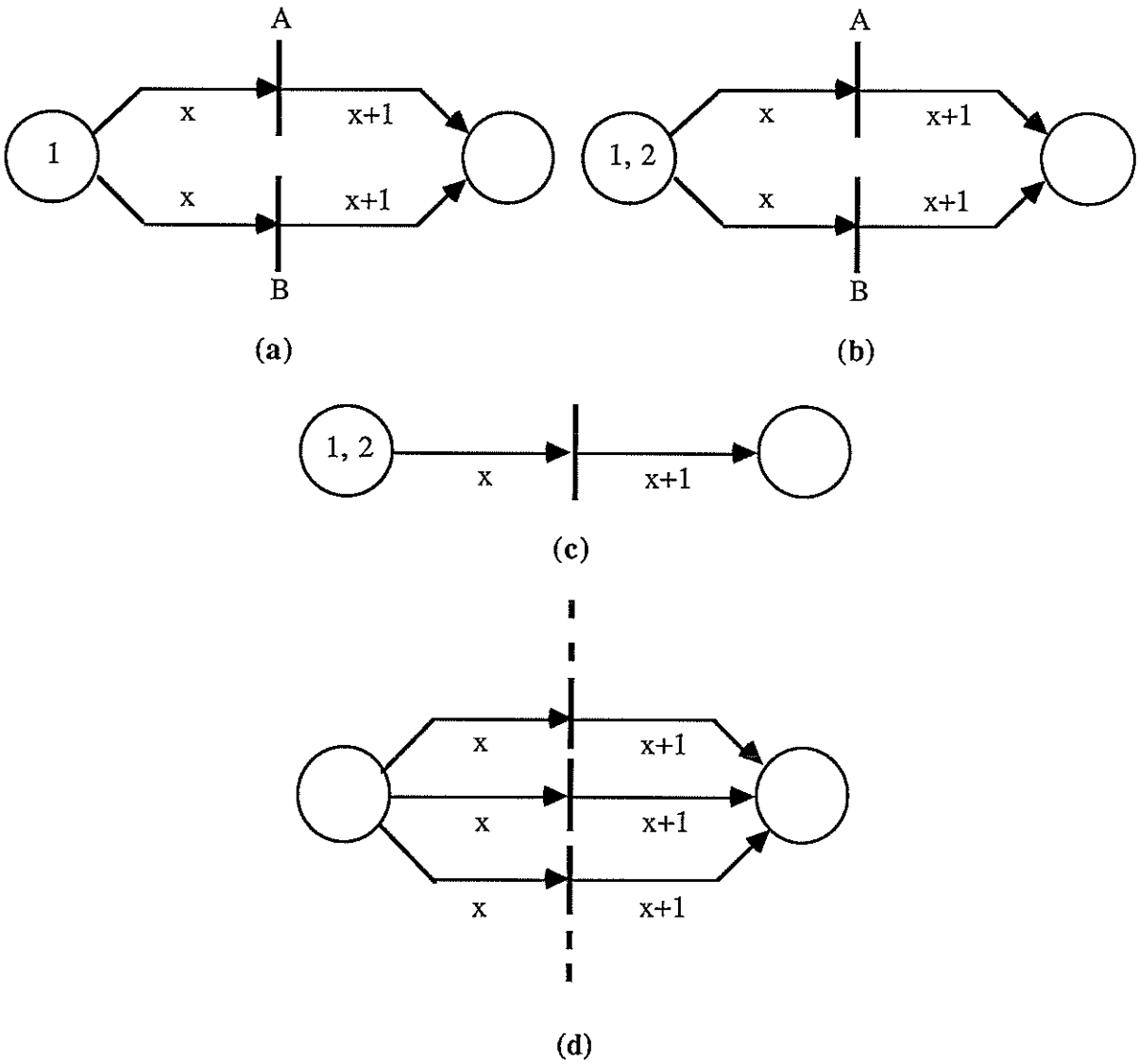

Figure 3: Concurrent Transactions

The net (a) is not concurrent because while A and B are enabled they are in conflict, i.e., the firing of one would block the firing of the other. On the other hand (b) is concurrent because the two transactions are enabled and no conflict exists. The net (c) is an interesting case. According to the above definition, this net is not concurrent because there is only one transaction. However, this net is isomorphic to (b) in the sense that any reachable marking of (b) is also a reachable marking of (c), and vice versa. As a matter of fact, it is easy to show that (c) is isomorphic to any net which has an unbounded number of the identical transactions between the same pair of databases as in (d). Therefore, by considering (c) as an abbreviation of (d), we claim that (c) is a concurrent net. Furthermore we will hereafter consider that every transaction box represents an unbounded, arbitrary number of identical transactions sharing the same source and target databases. With this understanding any net that contains more than one data object in some database is a concurrent net and the degree of concurrency can be measured in general by the number of objects in a database. In the following we will construct transaction net solutions for well known programming problems.

Concurrent Factorial (Figure 4): The nets presented in Figure 2 compute the factorial function sequentially. The net Figure 4(a) below generates n! into F in O(log(n)) time at best, but there is no guarantee for such performance due to the nondeterminism about pairing. (Note that we are assuming every transaction is duplicated in unbounded number of times.) In this net and the remaining examples, we will use the abbreviation ',' for ')(' in specifying queries and updates. For example, (x,y) ≡ (x)(y) and (u(x),v(y)w) ≡ (u(x))(v(y)w). However, there should not be any confusion about ',' used for separating two queries.

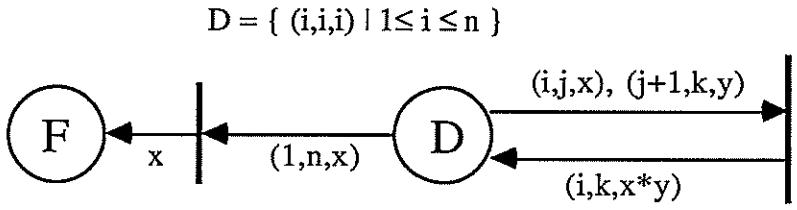

Figure 4 (a): Concurrent Factorial by Pairing

For example, one possible transformation sequence of D for computing 5! is as follows:

{ (1,1,1), (2,2,2), (3,3,3), (4,4,4), (5,5,5) }
{ (1,2,2), (3,3,3), (4,5,20) }
{ (1,2,2), (3,5,60) }
{ (1,5,120) }.

The net (b) also generates { (k, k!) | 1 ≤ k ≤ n } into F in O(log(n)) time using the prefix method [11]. It is assumed that n is a power of 2. A double-headed arrow abbreviates two arrows in each direction, i.e., the same objects consumed by the query will be produced back to the same database.

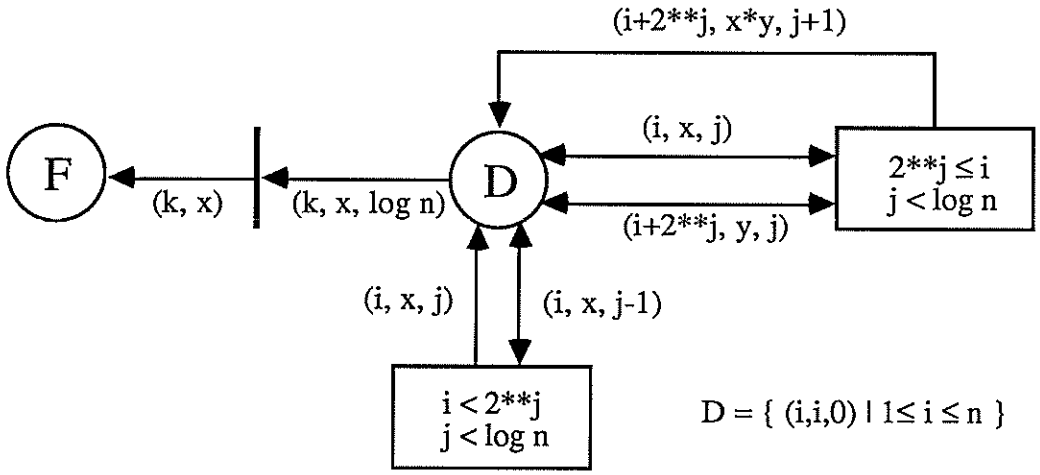

Figure 4 (b): Concurrent Factorial with Prefixing

The following condition holds at any time for the database D:

(i, x, j) ∈ D ≡ (x * (i - 2**j)! = i!) where n! ≡ 1 for any integer n < 1.

We now show that the condition holds for the initial state of D, and a firing of any transaction in the network preserves the condition; i.e., the condition is an invariant of the network. For the initial state: (i, i, 0) ∈ D ≡ (i * (i - 2**0)! = i!). Suppose that D satisfies the condition and the transaction on the right fires, producing (i+2**j, x*y, j+1) in D. We want to show that (x*y) * ((i+2**j) - 2**(j+1))! = (i+2**j)!, i.e.,

(x*y) * (i - 2**j)! = (i+2**j)!.

In order for the transaction to fire, there must be data objects, (i, x, j) and (i+j**2, y, j), in D such that 2**j ≤ i ∧ j < log n ∧ x * (i - 2**j)! = i! ∧ y * ((i+2**j) - 2**j)! = (i+2**j)!. Therefore, y * (x * (i - 2**j)!) = (i+2**j)!, i.e., (x*y) * (i - 2**j)! = (i+2**j)!.

Suppose that the transaction on the bottom fires, producing (i, x, j+1) in D. We want to show that x * (i - 2**(j+1))! = i!. In order for the transaction to fire, there must be a data object, (i, x, j) in D such that i < 2**j ∧ j < log n ∧ x * (i - 2**j)! = i!. Since i < 2**j < 2**(j+1), (i - 2**j)! = (i - 2**(j+1))!, therefore x * (i - 2**(j+1))! = i!. The transaction on the left does not produce any new data on D, hence no effects on the condition.

As a corollary of the above property the following condition holds for F:

(k,x) ∈ F ≡ (x = k!).

A computation of 8! (i.e., n = 8 and log n = 3) will generates the following set of data objects (i, x, j) into D:

	i = 1	2	3	4	5	6	7	8
j = 0	1	2	3	4	5	6	7	8
j = 1	1	2	6	12	20	30	42	56
j = 2	1	2	6	24	120	360	840	1680
j = 3	1	2	6	24	120	720	5040	40320

Sorting (Figure 5): For a given array $\{a_i\}$ of $n > 0$ numbers represented by a set of ordered pairs $A \equiv \{ (i, a_i) \mid 1 \le i \le n \}$, to construct the sorted array $\{b_j\}$ represented by another set of ordered pairs $B \equiv \{ (k, b_k) \mid 1 \le k \le n \}$ such that if $(i,x), (j,y) \in B$ and $i < j$, then $x \le y$, and $\{ a_i \mid (i, a_i) \in A \} = \{b_k \mid (k, b_k) \in B \}$.

In the sorting net of Figure 5, the database D initially contains a set of ordered pairs A and will contain B when the net terminates. A firing of the transaction reduces the number of inversions in D and the net terminates when no more inversion exists in D. Note that the box in the network represents unbounded arbitrary number of identical transactions.

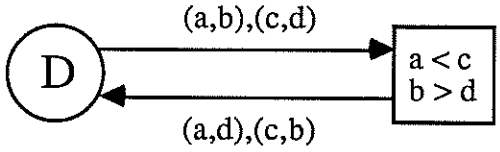

Figure 5: Sorting

Transitive Closure (Figure 6): Find the transitive closure R^+ of a binary relation R. Assuming that the initial database contains the set of ordered pairs representing R, the net in Figure 6 generates the transitive closure in the database. The net reaches the quiescent state because R^+ is finite when R is finite, and when the transaction tries to produce a pair (x, z) into R where it already exists, the transaction will be disabled according to the definition of the firing rule for a transaction.

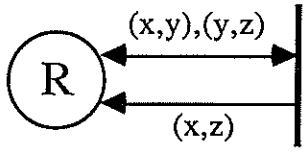

Figure 6: Transitive Closure

The Shortest Path (Figure 7): Find a shortest (minimum cost) path between a pair of vertices in a given directed graph R with the set of vertices $A = (a_1, a_2, \ldots, a_n)$, $n>2$, and the cost function $c: A^2 \rightarrow N$ such that for any $x,y \in A$,

$$\begin{aligned} c(x,y) &= 0 && \text{if } x = y \\ &= M && \text{if } (x,y) \notin R \\ &> 0 && \text{if } (x,y) \in R \wedge x \neq y \end{aligned}$$

where $M = n * \max\{ c(x,y) \mid (x,y) \in R \}$ is an upper-bound of the cost between a pair of vertices.

The net of Figure 7 computes the shortest paths for all pairs of vertices by updating the cost between the vertices x and z whenever a new path through y is found to be less costly. The invariant of the computation is that if $(x,y,c) \in DB$, then c is the shortest path length (the minimum cost) between x and y, known so far. When the net halts the following condition holds: For any vertices x, y, and z; the sum of the shortest path length between x and y, and the shortest length between y and z, is always larger than the shortest length between x and z.

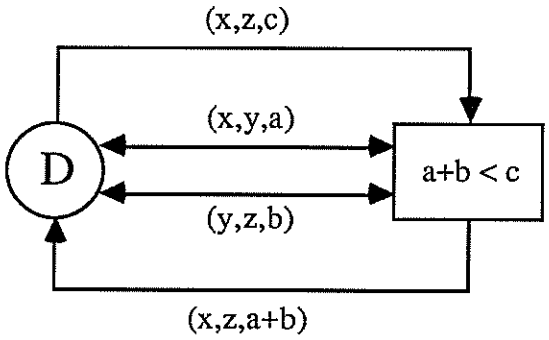

Figure 7: The Shortest Path Problem

Directed Hamilton Circuit (Figure 8): Given a directed graph R with the set of vertices A, to find a cycle containing every element of A.

In the net of Figure 8, it is assumed that A is represented by a single object $(a_1, a_2, \ldots a_n) \equiv (a_1)(a_2) \ldots (a_n)$, $n > 2$, stored in the database A, and R is represented by a set of ordered pairs $(a,b) \in R$, stored in the database R.

Note that the transaction Compute generates a permutation of $A \equiv (a_1)(a_2) \ldots (a_n)$ as follows: If $(x,y) \in C$, then xy is a permutation of A.

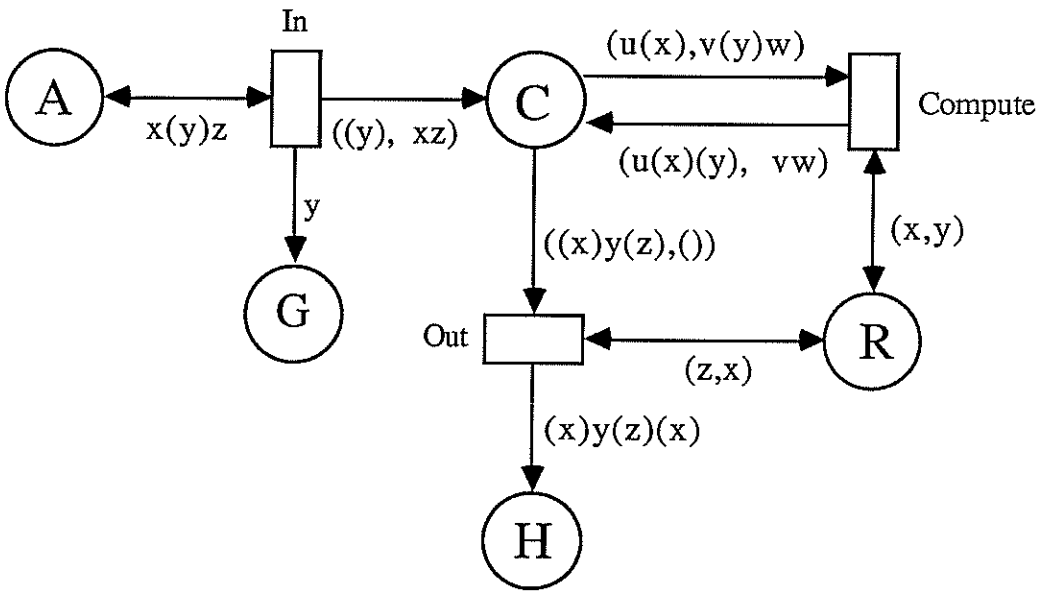

Figure 8: Directed Hamilton Circuit

To see how the algorithm works, consider: R={(1,2),(2,4),(2,3),(3,4),(4,1)} with A=(1)(2)(3)(4). The transaction IN nondeterministically produces into C the set of all possible starting and remaining vertices as { ((1), (2,3,4)), ((2), (1,3,4)), ((3),(1,2,4)), ((4), (1,2,3)) }. The database G guarantees that the transition IN would not produce redundant lists. The transaction Compute selects a next vertex from the right list and moves to the left, producing lists such as ((1,2), (3,4)) and ((3,4,1), (2)). The final output will be { (1,2,3,4,1), (2,3,4,1,2), (3,4,1,2,3), and (4,1,2,3,4) } in H.

The Eight Queen's Problem (Figure 9): Find all configurations of placing eight queens on the chessboard in such a way that no two queens attack each other.

We will represent a solution by a permutation of (1,2,3,4,5,6,7,8) satisfying the condition that $|a_i - a_j| \neq |i - j|$ for $1 \leq i < j \leq 8$. One such solution is (1,5,8,6,3,7,2,4). The net given in Figure 9 generates all possible permutations by sequentially testing the solution condition for each addition of new element into the candidate permutation. The initial database is ((), (1,2,3,4,5,6,7,8)).

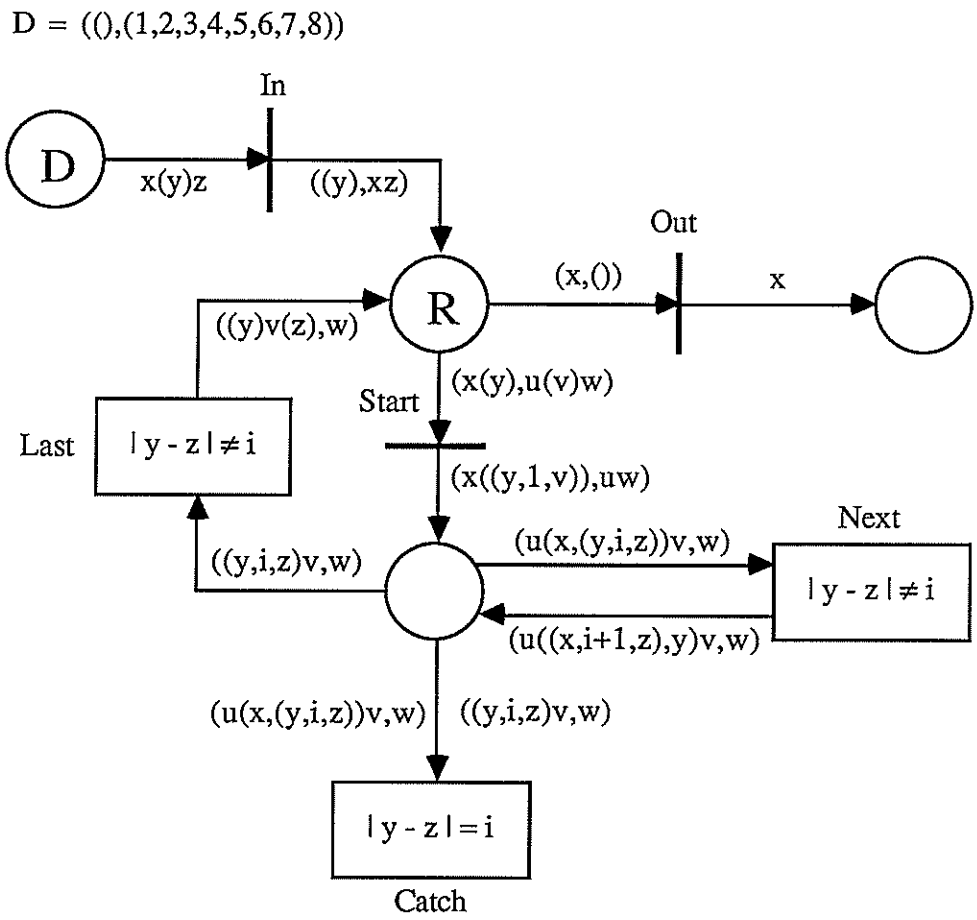

Figure 9: The Eight Queen's Problem

Of a list (x, y) in the database R, x represents a partially successful candidate permutation, and y represents a list of available remaining elements. For example, ((3,6,4),(1,2,5,7,8)) in the database shows that (3,6,4) is possibly a part of a solution for the eight queens problem. The transactions Start, Next, and Last select one element from the available pool non-deterministically and try to move it to the candidate list while testing the conflict with already selected candidates. For example, ((3,6,4),(1,2,5,7,8)) in R will be transformed into ((3,6,4,2),(1,5,7,8)) in R as follows:

((3,6,(4,1,2)), (1,5,7,8))	by Start
((3,(6,2,2),4), (1,5,7,8))	by Next
(((3,3,2),6,4), (1,5,7,8))	by Next
((3,6,4,2), (1,5,7,8))	by Last.

The above association will be completed into ((3,6,4,2,8,5,7,1), ()) in R, and then by Out one solution (3,6,4,2,8,5,7,1) will be generated. The purpose of the transaction Catch is to delete superfluous objects which failed the test.

4. Conclusions

As the examples in the previous section show the transaction net is a semantically simple, expressively powerful, visual, and concurrent programming language. Two major issues must be addressed before the concept becomes a significant alternative to the existing process oriented paradigm. One is the ease of programming. The other is the implementability.

The ease of programming in a particular language requires four major supports: software environment, simple semantics, friendly syntax, and a formal method of verification and derivation. Our next step in this area is to develop a proof theory for the transaction net. One possible approach is to construct a similar theory developed for the associon closure statements by Rem [7]. Another possibility is to translate a transaction net into a set of equations on the database states and to devise a method for finding their fixed-points as suggested by Chandy for the Unity project [1].

The implementability of a transaction net, in essence, can be reduced to that of large scale associative memory, i.e., content addressable memory, and to that of sharing such a resource by a large number of processors. There are two approaches for implementing the required mechanisms. One is to map those capabilities on the existing multiprocessor architecture such as the N-Cube, the Connection Machine, the Transputer, and the bus-oriented network of computers. The other is to design special purpose VLSI chips for large scale associative memory and for inter-processor communication networks. The Linda project, addressing the same implementation issue, is taking both approaches. Our next step will be to study the first approach with careful performance analysis.

5. References

[1] Chandy, M., Concurrent Programming for the Masses. Invited Address, 3rd Annual ACM Symposium on Principles of Distributed Computing, Vancouver, August 1984.

[2] Fikes, R. and Kehler, T., The role of frame-based representation in reasoning. CACM 28(9):904-920, September 1985.

[3] Forgy, C.L., The OPS83 Report, System Version 2.1. Production Systems Technologies, Inc. October 1984.

[4] Gelernter, D., Generative Communication in Linda. ACM TOPLS, 7:1 (1985), pp. 80-112.

[5] Holt, A. and Commoner, F., Events and Conditions.Applied Data Research, New York, 1970.

[6] Kimura, T.D., Choi, J.W., and Mack, J.M., Keyboardless Programming in Visual Language. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, June, 1986.

[7] Rem, M., *Associons and the Closure statements*. Mathematical Centre, Amsterdam, 1976.

[8] Peterson, J. L., *Petri Net Theory and The Modeling of Systems*. Prentice-Hall, 1981.

[9] Petri, C., Kommunikation mit Automaten. Ph. D. Dissertation, University of Bonn, West Germany, 1962.

[10] Rychener, M.D., Production Systems as a Programming Language for Artificial Intelligence Applications. Ph.D. Thesis, Carnegy-Mellon University, 1977.

[11] Stone, H.S., *Introduction to Computer Architecture*. Science Research Associates, Inc., 1976.

[12] Zervos, C.R., Colored Petri Nets: Their Properties and Applications. Ph. D. Thesis, University of Michigan, 1977.