Journal of Engineering and Applied Sciences 14 (13): 4295-4302, 2019
ISSN: 1816-949X
© Medwell Journals, 2019

Petri Net Based Event Driven Programming

Bahaa Mohsen Zbeel
College of Fine Art, University of Babylon, Babil, Iraq

Abstract: The idea of this research is how to program event driven systems such as graphical user interfaces, games which can be modeled with Petri nets, a graphical and mathematical modeling tool, using their incidence matrices and a suggested program structure. Incidence matrix defines and analyzes completely the dynamic behavior of Petri nets by some equations. In this way, a numerical method could achieve to adapt the program behavior easily with benefits of using the analysis power of Petri nets. The suggested program structure decomposes the overall program into 3 parts: the event handler's library part, the driver routine part and a resource file part containing the incidence matrix. In this manner the program reusability will increase and simplify the program adaptability process and shorten the program construction life time.

Key words: Models of computation, programming paradigms, reactive computation, reusable software, decomposes, adaptability process

INTRODUCTION

A wide variety of application, from high performance servers to enterprise applications to GUIs to embedded systems, rely on an event based programming style. Event driven programming implements a stylized programming idiom where programs use non-blocking operations and the programmer breaks the computation into fine grained callbacks (or event handlers) that are each associated with the completion of an I/O call (or event). This approach permits the interleaving of many simultaneous logical tasks with minimal overhead, under the control of an application level cooperative scheduler. Each callback executes some useful work and then either schedules further callbacks, contingents upon later events or invokes a continuation which resumes the control flow of its logical caller, the event-driven style has been demonstrated to achieve high throughput in server applications (Pai *et al*., 1999; Welsh *et al*., 2001), resources-constrained embedded devices (Gay *et al*., 2003) and business applications (Fischer, 2008).

Unfortunately, programming with events comes at cost event-driven programs are extremely difficult to understand and maintain. Each logical unit of work must be manually broken into multiple callbacks scattered throughout the program text. This manual code decomposition is in conflict with higher level program structuring. For example, calls do not return directly to their callers, so, it is difficult to make use of procedural abstraction as well as a structured exception mechanism.

Threads represent an alternative programming model commonly used to interleave multiple flows of control. Since, each thread maintains its own call stack, standard program structuring may be naturally used, unlike in the event-driven style. However, threads have disadvantages as well including the potential for race conditions and dead-locks as well as high memory consumption (Behren *et al*., 2003a, b). Within the system research community, there is currently no agreement that one approach is better than the other (Pai *et al*., 1999; Behren *et al*., 2003a, b; Adya *et al*., 2002). In addition in some contexts, threads either cannot be used at all (such as within some operating system kernels) or can only be used in conjunction with events (such as thread-pooled servers for Java servlets (Fischer, 2008). For more information about event driven programming paradigm (Salvaneschi *et al*., 2015).

In this study, a program structure is proposed depend solely upon Petri nets to model the event driven program by reinterpret the Petri nets components (places and transitions) to accommodate the event driven program paradigm.

The behavior of resultant program will be easier to be maintained that is due to the use of incidence matrix of Petri net which represents the Petri net-based program in numerical fashion this is in contrast with the classical imperative approach that uses an event handler after examining a triggering event that is acquired from events queue periodically using explicit instructions is much simple. All what the programmer needs to change the program behavior is to alter the incidence matrix entries and modify the callback functions related to it to accommodate any behavioral changes subjected to.

The proposed program structure isolates the callback functions from the main program (the driver routine) that

call them when an event trigger and leave them in a separate file that will called "the subroutine library". Due to this isolation and due to nature of the state equation (that uses the incidence matrix, control vector and marking vector to automate the Petri net model based program) which combines the expected events that may be occurred with the callback subroutine related to it the resultant program will be much easier to understand and more readable and more flexible for adaptation.

Petri nets preliminaries: Petri nets are a well established model of concurrent systems with a rich and strong, yet still growing theory. Petri nets are intensively used in the design, verification, analysis and prototyping of software systems, control systems and hardware systems. Different dialects or extensions of Petri nets serve in the different design, timed and stochastic Petri nets for performance evaluation (Badouel *et al*., 2015).

PN is defined as a bipartite weighted directed graph. it is composed of places, transitions, directed arcs and tokens. Places represent the conditions and transitions represent the events. Places and transitions are connected by directed arcs. These arcs represent the flow of events within the system. Tokens simulate the system dynamics. Places are used to specify the conditions whereas transitions are utilized to represent the triggering of events. Directed arcs connect the places and transitions together. It must be noted that same types of nodes cannot be connected. Namely, 2 places or 2 transitions cannot be connected together. Only different types of nodes can be connected (BaskocagIl and Kurtulan, 2011).

At any time of the evolution of a PN, places hold zero or a positive number of tokens. The state of the system is represented by this allocation of tokens over the places and is called a marking. The definition of a PN includes the specification of an initial marking which allocates a number of tokens to each place. A transition is enabled if its input places contain at least the required numbers of tokens (defined by the weight assigned to the arcs). The firing of an enabled transition will then result in the consumption of the tokens of its input places and the production of a number of tokens in its output places (this number is determined by the weights of the arcs going out of the transition) (Fig. 1). This "token game" represents the dynamical evolution of the system (Chaouiya, 2007).

A Petri Net (PN) is a construct $N = (P, T, F, \varphi)$ where P and T are disjoint finite sets of places and transitions, respectively, $F \subseteq (P\times T) \cup (T\times P)$ is the set of directed arcs, $\varphi: (P\times T)\cup(T\times P) \to \{0, 1, 2, ...,\}$ is a weight function where $\varphi(x, y) = 0$ for all $(x, y) \in ((P\times T) \cup (T\times P)) - F$. A Petri net can be represented by a bipartite directed graph with the node set $P\cup T$ where places are drawn as circles, transitions as boxes and arcs as arrows with non-negative integer labels. A Petri net N is called an Ordinary Net (ON) if $\varphi(x, y) = 1$ for all $(x, y)\in F$. We omit φ from the definition of an ordinary net. A mapping $\mu: P \to \{0, 1, 2, ...,\}$ is called a marking. For each place $p \in P$, $\mu(p)$ gives the number of tokens in p. ${}^{\bullet}x = \{y|(y, x)\in F\}$ and $x^{\bullet} = \{y|(x, y)\in F\}$ are called the sets of input and output elements of $x \in P\cup T$, respectively (Dassow and Turaev, 2009).

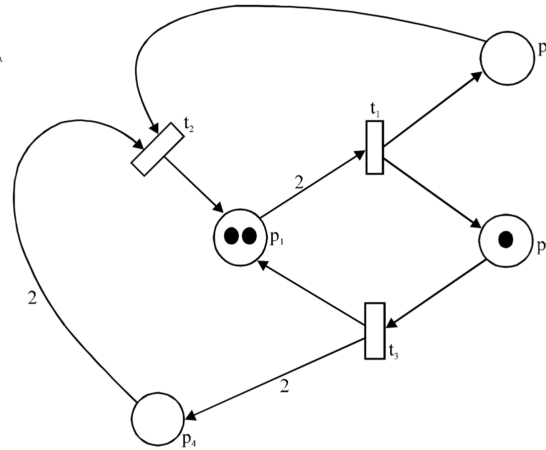

Fig. 1: A Petri net

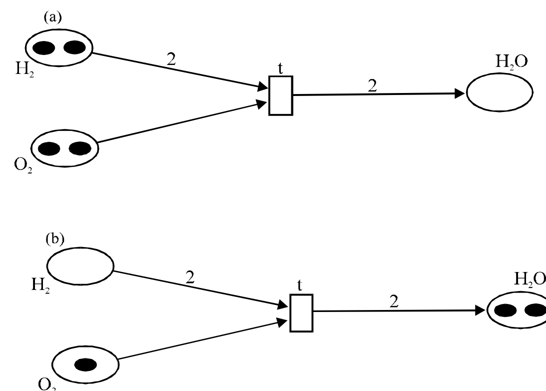

Fig. 2a): The marking before firing the enabled transition t and b) The marking after firing t where t is disabled

Example: The well known chemical reaction: $2H_2+O_2 \to 2H_2O$. Two tokens in each input place in Fig. 2a show that 2 units of H_2 and O_2 are available and the transition t is enabled after firing t, the marking will change to the one shown in Fig. 2b where the transition t is no longer enabled (Murata, 1989).

Incidence matrix and state equation: The incidence matrix of a directed graph G is a p×q matrix (a_{ij}) where p and q are the number of vertices and edges, respectively, such that a_{ij} = -1 if the edge ej leaves vertex v_i, (a_{ij}) = 1 if the edge e_j enters the vertex v_i and 0 otherwise (Awasthi, 2014). For a Petri net N with n transitions and m places, the incidence matrix $A = (a_{ij})$ is an n×m matrix of integers and its typical entry is given by:

$$a_{ij} = a^{+}_{ij} - a^{-}_{ij} \tag{1}$$

where, a^{+}_{ij} = w (i, j) is the weight of the arc from transition i to its output place j and a^{-}_{ij} = w (j, i) is the weight of the arc to transition i from its input place j (Murata, 1989). In writing matrix equations, we write a marking M_k as an m×1 column vector. The jth entry of M_k denotes the number of tokens in place j immediately after the kth firing.

The kth firing or control vector u_k is an n×1 column vector of n-1 0's and one nonzero entry, a 1 in the ith position indicating that transition i fires at the kth firing. Since, the ith row of the incidence matrix A denotes the change of the marking as the result of firing transition i, we can write the following state equation for a Petri net (Murata, 1977):

$$M_k = M_{k-1} + A^T u_k,\ k = 1, 2, ..., \tag{2}$$

For the Petri net shown in Fig. 2, the state Eq. 2 is illustrated below where the transition t_3 fires to result in the marking $M_1 = (3002)^T$ from $M_0 = (2010)^T$:

$$\begin{bmatrix} 3 \\ 0 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -2 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & - & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

MATERIALS AND METHODS

The suggested method steps are

Step 1: Reformulate the system description statementsinto a Petri net such that each place in it represents a system state (wait, process, print, insert, skip, etc.) and each transition represents an event that cause system state changes (paper jam, certain character read, deposit, stack overflow, temperature increase, end of file, etc.).

Step 2: Initially, mark the place that corresponds to the system start state that will call a start place with one token and assign initially 1 as a weight to each arc in the net.

Step 3: Change the start place initial marking and arc's weight until required net behavior is obtained.

Step 4: Extract the Petri net incidence matrix and put the matrix in separate file as program's resource file.

Step 5: Initialize control column vector u_k entries by 0 and initialize the initial marking vector, M_0 such that the start place initial marking is put in the start place position in the marking vector and zero in the other positions.

Step 6: Assign to each place a subroutine that play the role related to it (e.g., a subroutine "wait" in a communication protocol software is assigned to a "wait" place in a Petri net-modeled communication protocol, a subroutine "print" that manages the printing of document pages on a printer assigned to a "printing" place in a Petri net model that describes a text processing software and so on and put them in a separate file as library.

Step 7: Save the addresses of subroutines constructed in step 5 in a vector that will called Subroutine Address Vector (SAV) in the same order of occurrence of their corresponding places in the marking vector, M.

Step 8: Run the Petri net by reading an input (an event) and assign 1 in the control vector position that represent this event then applying 2. Here M_{k-1} for the first time is the initial marking, M_0.

Step 9: Run the subroutine assigned to the place that increased in its tokens.

Step 10: Repeat step 8 and 9, respectively until normal termination occur. The following diagram in Fig. 3 depicts the overall program structure according to the above steps.

Employing suggested method in games: Consider the following Petri net in Fig. 4, representing the "tom and jerry" game in which the cat "tom" is run after the mouse

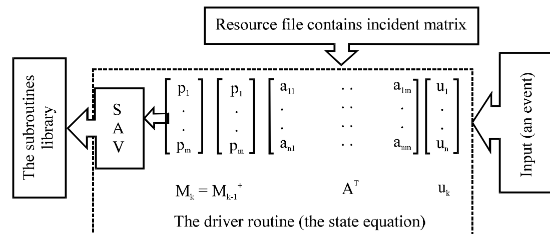

Fig. 3: The proposed program structure

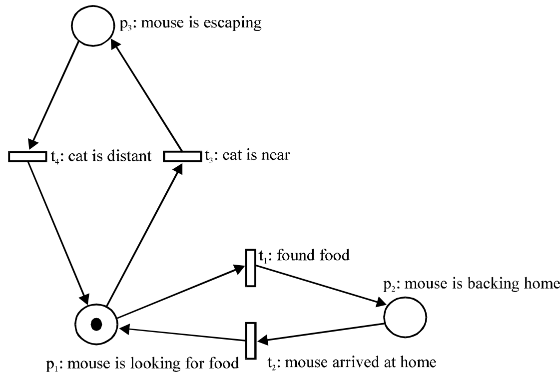

Fig. 4: "The tom and jerry" Petri net model

"jerry" which carrying food home, here, computer play the role of the mouse and the user is play the role of the cat. The incidence matrix of Fig. 4 is the model can run mathematically. The C" like pseudo code lists figures in the following pages is the overall program skeleton which consist of 4 parts present, respectively the initialization routine part, the driver routine part, the subroutines related to the Petri net places part and the user defined event part algorithm 1 and 2.

Algorithm 1; The initialization subroutine pseudo code:

```
Int intialization_part()
{
/*
the mouse_character header file included the subroutines
 which play the role of the places in the Petri net
These subroutines implemented as methods in a class
called "mouse_class". Essential Petri net configuration
 information such as the incidence matrix, the start
place number and number of tokens are found in
net_configration header file. */
#include<mouse_character>
#include<net_configration>
/*the following are the global data type and variables
  declarations.*/
mouse_class mo//mo short of "mouse object"
typedef void (*subroutine_address) (void)
typedef enum game_event {found_food
 mouse_arrived_at_home
 cat_is_near, cat_is_distant}
vector<subroutine_address>SAV
/*store subroutine's which play the role of
the Petri net places found in included libraries
addresses into subroutine address vector:*/
SAV.pushback(&mo.mouse_is_looking_for_food())
SAV.pushback(&mo.mouse_is_backinghome())
SAV.pushback(&mo.mouse_is_escaping())
// extract control vector length and define it
Control vector length = no_of_rows (Incidence matrix)
Vector <int>  control_vector (Control vector length)
//extract marking vector length, define and initialize it
Marking vector length = no_of_colmuns (Incidence matrix)
Vector <int>  marking (Marking vector length, 0)
/* intialize the start place using "number_of_tokens"
 found in net_configration header file.*/
Marking [1] = number_of_tokens
//determine the start place numerically as 1
Marked_place = 1
//now call the start place subroutine: mouse is looking for food
  SAV [marked_place] ()
Return 0
}
```

Algorithm 2; The driver routine pseudo code called when an event raise:

```
Int the_driver_routine (game_event event)
{
/* "Another_event_occures" is a boolean type
 property defined in the "mouse_class" used
 to stop/start current subroutine:*/
//stop current subroutine
Mo. Another_event_occures = true
/* initialize control vector with respect to the
 Event  triggered and current maraked place
 The expression will  type casting to the integer
 data type.*/
control_vector (1) = ( Marking [1] = = 1)*(event = = found_food)
control_vector (2) = (marking [2] = =1)*
                        (event = = mouse_arrived_at_home)
control_vector (3) = (Marking [1] = = 1)*(event = = cat_is_near)
control_vector (4) = ( Marking [3] = = 1)*(event = = cat_is_distant)
/*calculate the new marked place using the stEq function that
   perform the state equation*/
marked_place = stEq (marking, control_vector, incidence matrix)
//to start running the new subroutine
Mo. Another_event_occures = false
//call the new marked place subroutine
SAV [marked_place] ()
Return 0
}
```

The following Fig. 3 contain the subroutines pseudo codes, found as methods in mouse class which play the role of Petri net model of Fig. 4.

Algorithm 3; "Mouse is looking for food" subroutine pseudo code:

```
Void mouse_is_looking_for_food(void)
{
While! ( Another_event_occures)
{
/*
Place here the detailed code needed to implement
 this subroutine. In addition, check the expected
 events which maybe raise during running this
subroutine. The following two lines contain the two
events which may be triggered in this subroutine
*/
on_ found_food(event)
on_ cat_is_near(event)
}
}
```

Algorithm 4; "Mouse is backinghome" subroutine pseudo code:

```
Void mouse_is_backinghome(void)
{
While !( Another_event_occures)
{
/*
Place here the detailed code needed to implement this
Subroutine. and check the expected events which maybe
```

```
Raise during running this subroutine
The following line contains the event which may be
 triggered in this subroutine
*/
on_ mouse_arrived_at_home (event)
}
}
```

Algorithm 5; "Mouse is escaping" subroutine pseudo code:

```
Void mouse_is_escaping(void)
{
While !( Another_event_occures)
{
/*
Place here the detailed code needed to implement
 this subroutine and check the expected events
 which maybe raise during running this subroutine
The following line contain the event which may be
 triggered in this subroutine
*/
on_ cat_is_distant (event)
}
}
```

The events in pseudo codes found in algorithm 3-5 are user defined events defined in the "mouse-class" class. These events may be built on predefined events (such as change, paint or mouse movement events) of the container object of the game (main window layout of the game or its frame) and/or its child objects such as picture box or any other object. The general structure of these user defined events is in algorithm 6.

Algorithm 6; The user defined event general pseudo code:

```
Void event_name ( ) (game_event and event)
{/*Place here the detailed code needed to check if the
 necessary conditions are hold. If so assign to the
 "event" variable a proper event value (food_is_near,
 cat_is_near, mouse_arrived_at_home  or cat_is_distant
 ). For example, in the event on_cat_is_near, the event
 receive a boolean value "cat_is_approch" from the
 picture box (that contains the mouse picture) event
 "mouse movement". Note, "mouse cursor here repre-
sents the cat".*/}
```

Where "event-name" could be on-found-food, on mouse arrived at home, on cat is near or on cat is distant. The use of user defined events guarantees a level, respectively. The function "stEq" return the marked place index that contains number of tokens more than what was containing after applying (2). This index obtained by save the marking vector before calculating the state equation and during calculating the state equation applies the following expression for each of marking vector element: marked placed = marked place+((old marking [index]-marking [I])>0)*index). The sub expression (old-marking [index]-marking [i])>0) is a boolean expression returns 1 if evaluated as true, 0 otherwise. In this way, the "marked-place" variable which initialized to zero before calculating the state equation, always saves the index place with this property. The determination of marked place in this manner is useful when more than one place is marked after calculating a state equation as in the situation that will discuss in Fig. 5.

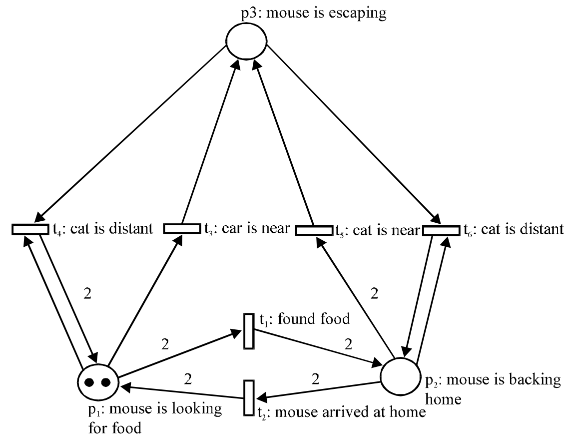

Fig. 5: "The updated tom and jerry" Petri net model

RESULTS AND DISCUSSION

The suggested method advantages are: The suggested method closes the gap between Petri nets based software modeling and code based on it by using the interpretation for places and transitions demonstrated in suggested method steps and by using the state equation as automation mechanism.

Ease of adaptability: The suggested method simplify the adaptation of the behavior of the resulted program without extra data structure and with few code changes. To understand how the suggested method will ease the program adaptability, consider the following situation: since, there are transitions connecting the place (mouse is looking for food) and the place (mouse is escaping), the mouse during its looking for food will always escape from the cat when it approaches. That will not occur if the marked place is (mouse is backing home), review the algorithm 1. In such situation the mouse return to its nest carelessly, no moving to the (mouse is escaping) place from (mouse is backing home) place. Suppose that the mouse also needs to escape from the cat when it is backing home. The Petri net model in Fig. 4 can be updated to the following model depicted in Fig. 5.

Algorithm 7; Updated tom and jerry Petri net control vector assignment:

```
Control vector [1] = (Marking [1] = = 2)* (event = = found _food)
Control vector [2] = (Marking [2] = = 2)* (event = = mouse_arrived_at_home)
Control vector [3] = (Marking [1] = = 2)* (event = = cat_is_near)
Control vector [4] = (Marking [3] = = 1)* (Marking [1] = = 1) && (event = = cat_is_distant)
Control vector [5] = (Marking [2] = = 2)* (event = = cay_is_near
Control vector [6] = (Marking [3] = = 1)* (Marking [2] = = 1) && (event = = cat_is_distant)
```

The transition (cat is near) in Fig. 5 which connect the place (mouse is backing home) with the place (mouse is escaping) is the solution for the above situation. But the problem is how can the mouse return to the last state was in before running away which may be “mouse is looking for food” or “mouse is backing home” place?. To solve this problem the Petri net model of Fig. 5 keeps one token either in “mouse is looking for food” or in “mouse is backing home” places after firing the “cat is near” t. or t. transition. So, the right “cat is distant” transition t. or t. will be enabled to firing depending on which of the 2 places “mouse is looking for food” or “mouse is backing home” is marked, therefore, the mouse will return to the last state was in it before escaping from the cat. The incidence matrix of Petri net in Fig. 5 is:

$$\begin{bmatrix} -2 & 2 & 0 \\ 2 & -2 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

According to model of Fig. 5, the main change to the program will be 1 changing the start place initial marking by assigning 2 to the “number of tokens” constant found in net configration header file, 2 changing the control vector initialization in the driver routine with respect to the newly added events initial marking distribution and current marked place as in Fig. 6.

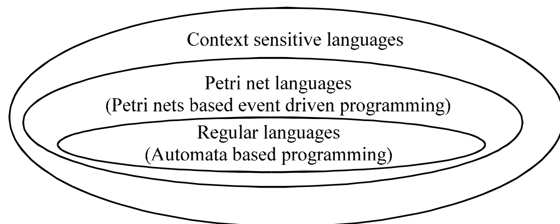

Fig. 6: Petri net based event driven programming vs. automata based programming

The incidence matrix that govern the behavior of the program can be constructed automatically using learning based computational structure such as neural nets with supervised learning (here input to the neural net is the events and the desired output are the subroutines to be run related to these events) or evolutionary algorithms such as Genetic algorithm (the initial population is number of incidence matrices their entries contain random values).

The suggested method is suitable for programming using visual programming technique that offers predefined events for a set of reusable objects (components) and offers the ability of constructing a user defined events for a user defined objects.

According to Chomsky Hierarchy, the suggested method that use Petri nets as a main theory for programming can be used to code tasks that cannot be coded in automata based programming which is use a finite states automata as a theory for programming. This is due to the fact that finite state machine can recognize just regular languages (the set of all events strings that describe the behavior of the task to be coded) which is a subset of Petri net languages which in turn subset of context sensitive languages. It has been shown that all Petri net languages are context sensitive languages (Peterson, 1981). Figure 13 is depicts this advantage.

In comparison with other approaches based on other computational models such as finite state automata, the adaptation of Petri net based event driven program needs lower change requirements as seen above. In contrast, the adaptation for the automata based program version of the “tom and jerry” game that equivalent to the model depicted in Fig. 4 to behave like the model in Fig. 5 require use of “the stack” data structure (this is mean using push-down automata) to realize the change needed with all the programming overhead of using this data structure from its programming requirements viewpoint (the subroutines needed to manipulate it such as push item in it, pop item from it, overflow checking, underflow checking and memory management related to it). By Peterson (1981) shows that the size of the state space of a push-down automaton, speaking a context free (type 2) language, grows exponentially with the length of the input string. However, the state space of a Petri net grows only combinatorially with the length of the input string. Clearly, the larger state space of the push-down automata allows them to represent languages that cannot be represented by Petri net. However, at any given time, a push-down automaton has access only to the current input and the top element of the stack. A Petri net, on the other hand with its comparatively complex interconnections between places and transitions has access to a large number of

counters at any given time this allows the Petri net to represent languages that cannot be represented by a push-down automaton, despite the smaller state space size.

The resulted Petri net based program model has a wide spectrum of properties that can be studied using Petri net theory such as reachability, boundedness, liveness, reversibility and home state, coverability, persistence, synchronic distance, fairness and completely controllable besides number of analysis methods such as coverability tree method, matrix equation approach and reduction or decomposition techniques. All of them, from software engineering viewpoint can be used as a formal method for program design and analysis algorithm 8.

Theoretically, Petri net languages are a context sensitive languages. If Petri nets used to model programs it could be described as context sensitive grammars for generating context sensitive languages which represent the behavior of the modeled programs where terminals symbols represent (events and subroutines names) and non terminals symbols represent (program states). For example, the following grammar describes the program behavior derived from updated "tom and jerry" Petri net model:

$$S \rightarrow e\ a\ B \mid e\ c\ E \mid e$$
$$e\ c\ E \rightarrow e\ c\ g\ d\ S$$
$$B \rightarrow f\ b\ S \mid f\ c\ E \mid f$$
$$f\ c\ E \rightarrow f\ c\ g\ d\ B$$
$$E \rightarrow g$$

Algorithm 8; Behaviour derived from updated "tom and jerry" Petri net model:

Where:
S = "Mouse is looking for food state" is the start symbol
Σ = { a = "found_found event"
b = "mouse_arrived_at_home event"
c = "cat_is_near event"
d = "cat_is_distant event"
e = "mouse is looking_for_Food () subroutine"
f = "mouse is backing_home () sub routine"
g = "mouse is escaping () subroutine"
} is a finite set of terminals
V = { B = "mouse is backing home state"
E = "mouse is escaping state"
}is a finite set of non terminals

The gain here is the software engineer can analyze the Petri net based program model using formal language theory and can answer some important questions such as (is certain string of events that lead to critical state belong to the language that describes program behavior?) this question is called the membership problem and this is very useful in the situations such as when programs control a system in environment where people life or any valuable entity must be protected or checked. Another application of formal theory for Petri net based programs occurs when the software engineer want to optimize the current Petri net based program model by deciding if another Petri net model represent the required program with less places or arcs is equivalent to the current one by comparing their formal languages (the set of strings generated by all firing sequences) if these 2 languages are equal then the new Petri net model is more optimal than the current one.

CONCLUSION

From software engineering viewpoint, the incidence matrix can be viewed as software Control Specification (CSPEC) which can be used as program behavior map enables the software engineer to adapt the program behavior easily. This will minimize the software adaptation cost and increase the whole software reusability by reuse all or part of incidence matrix which represents the Petri net model based program for applications with similar behavior, for example, we can reuse the incidence matrix of "tom and jerry" game illustrated above to build a military game that contains 2 players where the mouse player is replaced with a reconnaissance plane explores (the mouse is looking for food in tom and jerry game) the back lines of the enemy forces and then returns to the base (mouse is backing home in tom and jerry game) of its launch and the role of the cat is replaced with fighter jet defense chasing the reconnaissance aircraft.

Because the software user initially knows only the overall required software behavior, a set of externally observable software running mode which called states and the state is the main component of Petri nets, then the software engineer can use Petri net as an initial software design derived easily from the user requirement statement and after careful reviewing with the user the software engineer can obtain a high level software design which can be stepwise refined in a top down fashion, to obtain a detailed Petri net model for the required software which can converted to source code, of course after completing other analysis and design aspects, according to the suggested method. In this way, a speeding up can be obtained by merging the behavioral analysis and modeling phases in 1 phase and the procedural abstraction, due to the use of the stepwise refinement mentioned here in event driven programming can achieved easily.

SUGGESTIONS

A major weakness of Petri net is the complexity problem, i.e., Petri net based models tend to become too large for analysis even for a modest size system (Murata, 1989). A layered design approach is suggested to overcome this problem by using hierarchal Petri net model in which each state is a subsystem or a subprogram that can be reformulated and programming using the suggested method and continue in this manner in top down fashion. This is accomplishing by merging in one place each place related to same object (such as printer ready, paper jam and printer is busy places in a word processing software model that belong to the printer object) or occurs successively frequently (such as mouse is looking for food and mouse is backing home places in “tom and jerry” game modeled in Fig. 5 and expand it in the next layer. In this way, the complexity problem is controlled by keeping only the main places (states) that describe the overall current layer, therefore, the analysis of the model at each layer will be simplified.

RECOMMENDATIONS

Future research will focus on finding the more reasonable software process paradigm for employing the suggested method in a way that guarantee creating a timely, high quality software.

ACKNOWLEDGEMENT

I would like to extend my thanks to all researchers that I have adopted their research studies or dissertation as sources for this study for their contributions in supporting the scientific movement. I would like also to thank particularly both Prof. Jurgen Dassow and Taishin Yasunobu Nishida for answering my questions about Petri nets languages.

REFERENCES

Adya, A., J. Howell, M. Theimer, W.J. Bolosky and J.R. Douceur, 2002. Cooperative task management without manual stack management. Proceedings of the Annual International Conference on General Track of the USENIX Annua Technical, June 10-15, 2002, USENIX Association Berkeley, California, USA., ISBN:1-880446-00-6, pp: 289-302.

Awasthi, V.V., 2014. A note on the computation of incidence matrices of simplicial complexes. Intl. J. Pure Appl. Math., 90: 433-438.

Badouel, E., L. Bernardinello and P. Darondeau, 2015. Petri Net Synthesis. Springer, Berlin, Germany, ISBN:978-3-662-47966-7, Pages: 339.

BaskocagIl, C. and S. Kurtulan, 2011. Generalized state equation for Petri nets. Wseas Trans. Syst., 10: 295-305.

Behren, J.R.V., J. Conditand and E.A. Brewer, 2003a. Why events are a bad idea (for high-concurrency servers). Proceedings of the 9th International Conference on Hot Topics in Operating Systems HOTOS'03 Vol. 9, May 18-21, 2003, USENIX Association Berkeley, California, USA., pp: 19-24.

Behren, R.V., J. Condit, F. Zhou, G.C. Neculaand and E. Brewer, 2003b. Capriccio: Scalable threads for internet services. Proceedings of the ACM International Conference on SIGOPS Operating Systems Review Vol. 37, October 19-22, 2003, ACM, New York, USA., ISBN:1-58113-757-5, pp: 268-281.

Chaouiya, C., 2007. Petri net modelling of biological networks. Briefings Bioinf., 8: 210-219.

Dassow, J. and S. Turaev, 2009. Petri net controlled grammars: The case of special Petri nets. J. Univ. Comput. Sci., 15: 2808-2835.

Fischer, J.M., 2008. Robust Service Composition. University of California, Los Angeles, California, Pages: 261.

Gay, D., P. Levis, R.V. Behren, M. Welsh and E. Brewer *et al*., 2003. The nesC language: A holistic approach to network embedded systems. Proceeding of the 2003 ACM International Conference on Programming Language Design and Implementation (PLDI), June 9-11, 2003, ACM, San Diego, California, USA., pp: 1-11.

Murata, T., 1977. State equation, controllability and maximal matchings of Petri nets. IEEE. Trans. Autom. Control, 22: 412-416.

Murata, T., 1989. Petri nets: Properties, analysis and applications. Proc. IEEE., 77: 541-580.

Pai, V.S., P. Druscheland and W. Zwaenepoel, 1999. Flash: An efficient and portable web server. Proceedings of the USENIX Annual Conference on Technical General Track, June 6-11, 1999, USENIX, Monterey, California, USA., pp: 199-212.

Peterson, J.L., 1981. Petri Net Theory and the Modeling of Systems. Prentice-Hall, Upper Saddle River, New Jersey, USA., ISBN:9780136619833, Pages: 290.

Salvaneschi, G., A. Margaraand and G. Tamburrelli, 2015. Reactive programming: A walkthrough. Proceedings of the 2015 IEEE/ACM 37th International Conference on Software Engineering (ICSE'15) Vol. 2, May 16-24, 2015, IEEE, Florence, Italy, ISBN:978-1-4799-1934-5, pp: 953-954.

Welsh, M., D. Cullerand and E. Brewer, 2001. SEDA: An architecture for well-conditioned, scalable internet services. Proceedings of the ACM International Conference on SIGOPS Operating Systems Review Vol. 35, October 21-24, 2001, ACM, New York, USA., ISBN:1-58113-389-8, pp: 230-243.