SHARED-TRANSITION PETRI-NET BASED SIMULATION FOR IMPLEMENTATION IN A UNIVERSAL MACHINE CONTROL ENVIRONMENT

Saparudin Ariffin
R.H. Weston
and
R. Harrison
MSI Research Institute,
Department of Manufacturing Engineering
Loughborough University of Technology

ABSTRACT

This paper presents a modular Petri-net (MPN) based approach to modelling and analysing machine control processes. Here production lines which are characterised by a set of modules (or machine objects) can be controlled by a MPN. Interaction between different modules and workpieces is encoded within the Petri-net by so called shared-transition links. The paper describes the integration and implementation of modular Petri-net within a Universal Machine control (UMC) software environment which has previously been devised and produced within the MSI Research Institute at Loughborough University. Potentially the combined used of Petri-net and UMC offers new and exciting opportunities to optimize the behaviour of concurrent systems and to enable application in industrial machine control.

1.0 INTRODUCTION

In contemporary manufacturing enterprises, computer systems have numerous uses which include machine control. Common resource elements used within a manufacturing enterprises, which include various production machines, robots and transport devices. These elements need to be controlled in a flexible way (whether by people or computers) thereby promoting their reusability. Typically workpieces are required to flow between resource elements so that appropriate product realisation processes can be carried out. When employing a computer to control a machine system capabilities for processing a family of workpieces can be included, this potentially making such resources re-usable building blocks of a flexible manufacturing enterprise. Hence increasingly commonly multiple jobs share a set of reusable resources. However in such systems (as with other physical systems involving concurrency and the need for synchronism), problems of cònflict and deadlock can occurred. to help overcome such problems Petri-net specification techniques can be employed to schedule workflows; as Petri-nets have the ability to represent concurrency and conflicts among tasks. Petri-nets are effectively formal, graphical modelling tool capable of describing concurrent and distributed systems (Yim and Barta 1994) and as such are finding use in the design of computer software and hardware, communication systems and manufacturing systems. However, use of the basic Petri-net formalism can result in major constraints when handling large complex systems. As the number of Petri-net objects is increased, modelled systems become unmanageably complex making them very difficult for humans to interpret and manipulate; thus under such circumstances it extremely difficult to make modifications if the solutions defined need to be changed or extended.

2.0 A TASK LEVEL SIMULATION MODEL

The goal of the first author's research study is to produce a design concurrent for machine control based on the use of selected software architectures and formal specification techniques; and to implement the designs so derived by using the Universal Machine Control (*UMC*) software environment. A functional type of software architecture was selected and analysed as described by Ariffin et al (1994). The main advantages are the combined used of a functional architecture and the *UMC* program development and implementation environment include: the reusability of existing code and functions; support of a modular programming method; easier extension of existing programs; and the flexibility and extendibility of the approach to support various application domains.

In such a schema, the basic motion planning and monitoring algorithms used to control workflows through resources need to be modelled to facilitate design supported by simulation and visualisation of candidate solution. For example, such models should characterise the behaviour of machines used (e.g. class, attributes and parameters and the workflows involved, e.g. states, velocity and acceleration).

If good models of the system can be obtained it then becomes possible to use computer simulation to establishing and optimize the control logic of a machine system. Figure 1 illustrates typical procedural steps involved in engineering a machine system using the authors approach. When the basic control logic is determined, analysis and description of a particular operating sequence for a machine system can commence.

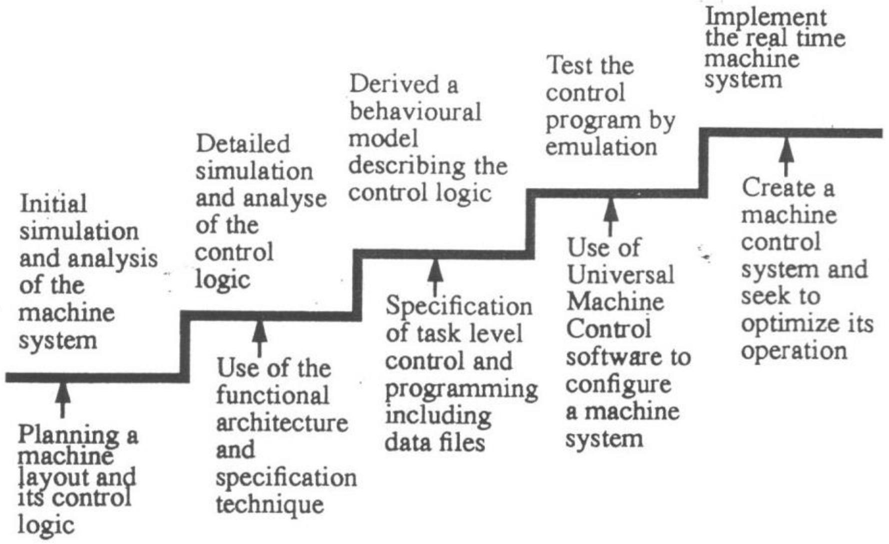
Initial simulation and analysis of the machine system
Planning a machine layout and its control logic
Detailed simulation and analyse of the control logic
Use of the functional architecture and specification technique
Derived a behavioural model describing the control logic
Specification of task level control and programming including data files
Test the control program by emulation
Use of Universal Machine Control software to configure a machine system
Implement the real time machine system
Create a machine control system and seek to optimize its operation

Figure 1 : Procedural Steps Involved in Designing and Implementing Task Level Control of Machine Systems

3.0 SIMULATION OF HIERARCHICAL SYSTEMS USING PETRI-NETS REPRESENTATIONS

The ability to represent the decomposition of a design in hierarchical form is a basic requirement for a system designers (Dotan 1991). Hierarchical representation can naturally support a top down design methodology. Various hierarchical manufacturing control architecture concepts were developed by Albus et al (1989) and Johnson (1990).

passes information to another parallel process. Thus the need for inter process communication and synchronization mechanism.

When a model in constructed it should have an inherent ability to characterise the behaviour of a system under variety of operating conditions (Wiendahl et 1991). Based on a stochastic Petri-net modelling approach it is possible to examine and optimize real processes and the behaviour of real systems (Zhou et al 1990). In such an approach a model describes time dependent processes and interactions within the real system. By analysing the timing relationships using a Petri-net specification technique, each task can be optimized by choosing the most appropriate physical components and at the same time avoiding deadlocks between processes.

Petri-net models can thus function as virtual hardware and as such can be used to simulate how the system will actually behave (Krogh and Genter 1990). These models can be utilized in design and operating phases of machine control systems. Especially for design, a full description of the states, events, time delays and activities of hardware components can be provided with Petri-net objects that are transparent to practical users. In the operating phase, the Petri-net model can be used as an emulator. At this stage, the current state of a system can be mapped to the Petri-net model by appropriate assignment of initial tokens.

A software tool can be developed to support the specification technique of Petri-nets. The output from such a tool can display and verify the correctness of connection between places and transitions, and the time-stamp of each place. The movement of tokens and firing sequences can be displayed to simulate and visualise conditions that will occur in a machine environment. Here control logic is required to drive the concurrent operation of Petri-nets, including the synchronization protocols required for Petri-nets using distributed processes.

Once a model of machine system has been created it may run as a simulation. Task behaviour is emulated by the generation of events for every node of the model. A node represents for example an assembly or buffer station and is described by a list of its properties including a time stamp. An event is properties including a time stamp. An event is simply a statement that information or control is transferred for processing at some other node at a specified time delay. To be closer to the way that process dynamics occur in real systems, it should be possible for several events to occur in real systems, it should be possible for several events to occur in parallel. In other word, several transitions should be allowed to fire simultaneously if they do not share resources. Furthermore, an event-generated task is normally repeated until a predefined stop criteria has been reached. Under such conditions a Petri-net simulation should faithfully follow the behaviour of the described system, this being characterized by transition's firing and the movement of tokens in the places. When all input places with tokens are searched and found, the relevant transition can be enabled and instantiated and firing of the transition can start. The flow of events occurring in a concurrent system is similar to the computation process which take place when the execution of fired transitions occurs in a simulation program. An event (transition) is enabled (fired) when all the preconditions (tokens in input places) are fulfilled.

The Petri-nets simulation model is separated into hardware and control systems. This is to achieve a closed similarity between a simulation model and a real system. In other words, the modelling process becomes more realistic and is more easily accomplished. In this way, the implementation of control logic is easily performed by simply converting the control logic in the simulation model. While simulation allows a designer to verify the system performance for a specific system, algorithmic-based analysis of Petri-net models has the potential for verifying the completeness and consistency of a design with respect to all possible real systems (Krogh and Genter 1990).

Using the methods developed by the authors, the simulation program is then incorporated into *UMC* environment. The program is first complied in a *UNIX* environment which is then forked by the *UMC* configuration editors. After setup and instantiation of the task program, the execution result is displayed in a *UMC* task window. The *UMC* configurations support the handling of multi-tasking and inter process communications (*IPC*). There are three type of *IPC* in *UMC*; these include: events; signal; and user data modules. All these *IPC* methods are based on use of the OS-9 operating system. The approach to compilation and execution is illustrated in Fig. 2. The OS-9 configuration editors, *UMC* task window and *UNIX* text editors exists in the same computer environment.

5.0 MODULAR AND SHARED-TRANSITION PETRI-NETS

5.1 The Concept of Shared-Transition Petri-nets

The concept of Modular Petri-nets (*MPN*) was proposed by Ariffin and et al (1995) and analysed. Such nets exhibits properties of liveliness within a confined module. the most important criteria is to have a manageable and easily extendible Petri-net system which functions as a virtual hardware analogue of real systems. In the proposed *MPN*, flexibility when modelling can be achieved by defining a new Petri-net module to represent any addition or modification of hardware components. The communication between Petri-net modules is accomplished by creating shared-transitions. In a real

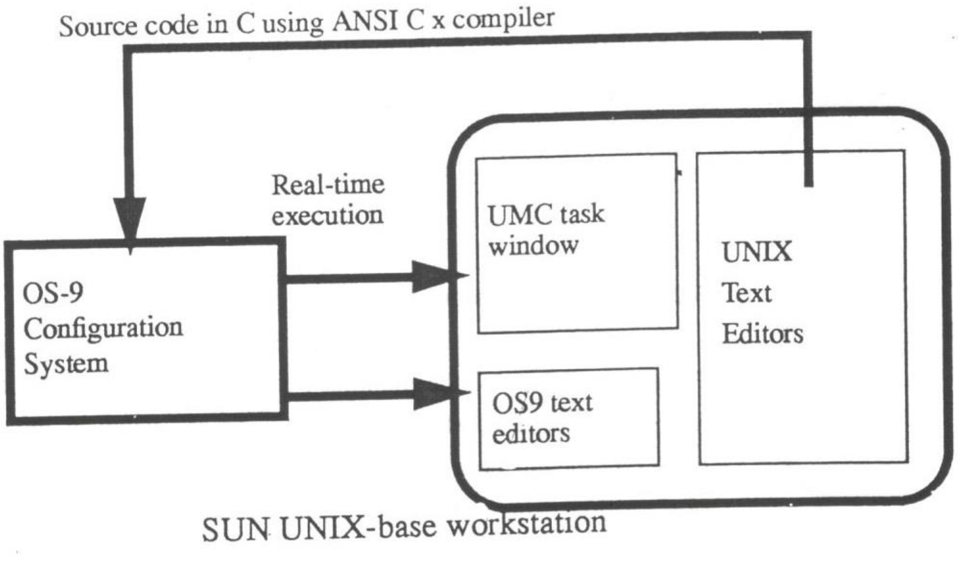
Source code in C using ANSI C x compiler
Real-time execution
OS-9 Configuration System
UMC task window
UNIX Text Editors
OS9 text editors
SUN UNIX-base workstation

Figure 2 : The Hardware and Software Environments

system, this takes the form of a communication path or connecting link between physical resources or hardware components.

A piece equipment or group of equipment items can be considered as one task or module. The Petri-net graph within the task is represented as a network of communicating processes. Intertask communication between at least two resources must exist in order to carry out the necessary job. This fact can be explained clearly by examining a simple machine system layout as shown in Fig. 4. The machine cell consist of a robot, an input conveyor and a processing machine. The robot unloads them on the machine. This operation can be represented by the modular and shared-transition type of Petri-net as shown in Fig. 5. All single-transitions connected to a shared-transition must be enabled for firing before the shared-transition could then be enabled. A transition concerns a dynamic properties of the system and represents a state change to some other state. a shared-transition (*ST*) represents communication of certain state conditions between at least two hardware components, such as when a job or part is transferred from one machine to another machine. For example, as seen from Fig. 5, *ST1* represents a part being transferred (picked-up) from the input conveyor by the robot manipulator and *ST2* represents the part being place down into the machine. In order for a transition to fire, each of the incoming places must be marked by tokens: a necessary condition for a transition to be enables. the tokens represented by dots show the flow of objects in machine systems. Each token belongs to certain object type such as a part, pallet or machine. After firing, the tokens are transferred to their respective outgoing places.

5.2 Time Extension

Having implemented the modular Petri-net schema for each task, an attribute of time is assigned to tokens, which will determine the delay in the token's flow. Each token carries a time attribute that is stamped at the moment of token's creation or at each instant of transition firing. A single-transition delay always updates the time attribute thereby maintaining latest time-stamp on corresponding incoming tokens. The time delay for a shared-transition is based on the maximum time-delay of all relevant incoming single-transitions (*T*). The model implemented in this shared-transition net is modular, which means any change in one part of the net cycle does not affect the rest of it. This approach also supports better understanding of the real mechanisms in a system via a distinct execution of part of the net cycle.

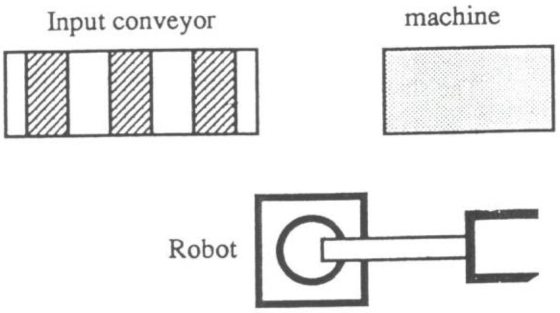

Fig. 3 A Simple Robotic Cell Layout

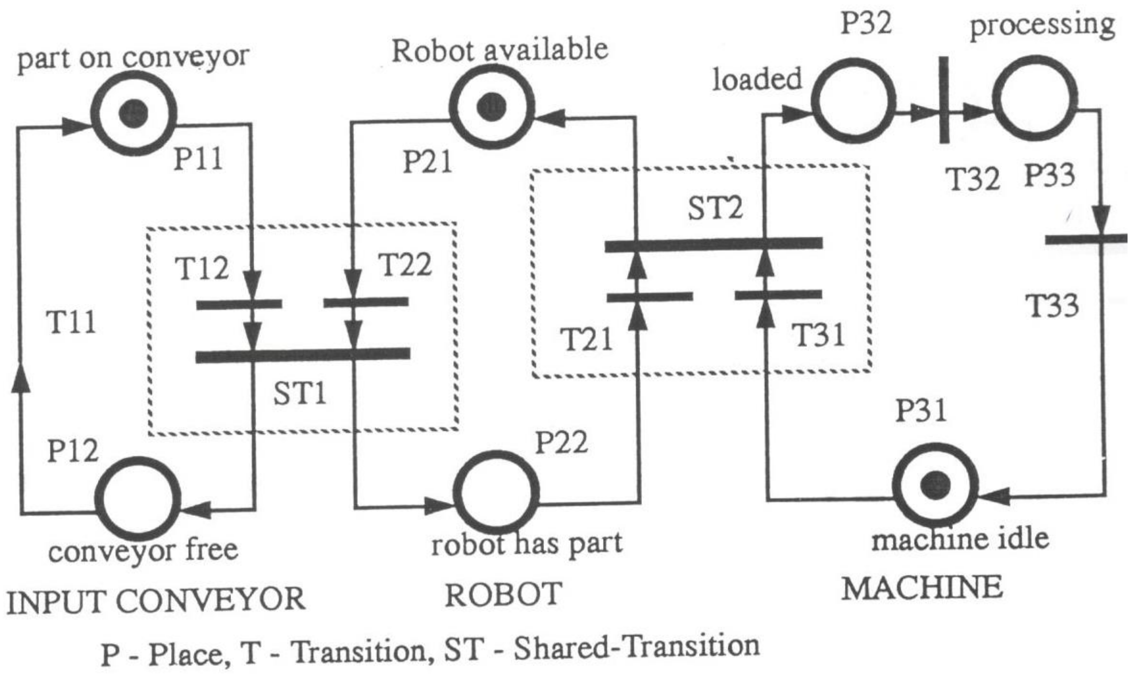

Fig. 4 A Shared-Transition and Modular Petri-net Model

5.3 Shared-Transition Algorithm and Programming Method

The shared-transition algorithm is selected so that it is easy to accommodate any change, especially with respect to data file and software development. In this study the building of data structures in a source code file is considered to be important. Figure 4 illustrates how the formation of a datafile can be used as input into a source code file, which following execution or simulation of this source code can result in an output file or record.

The programming algorithm starts by forming an input datafile (to read from) and an output file (to write). In then builds a read function by inputting all Petri-nets parameters used to model a machine system. The simulation starts by checking the status of all single transitions (*T*) whether they are enabled or not. In order for a transition to be enabled, all input places must be marked 'ON'. This is followed by checking if the shared transitions (*ST*) are enabled. This is done by establishing that all attributes of transitions are enabled in the first place. As soon as the *ST*s and *T*s are enabled, firing statements for the transitions are displayed, followed by unmarking of the marked places and marking the next places to indicate token movement. The status of *ST*s and *T*s must also be put into the 'OFF' condition. However, before a firing statement is displayed, the current time stamp at relevant places must be calculated. The output time stamp from a fired *ST* or *T* must be carried over to subsequent transitions before firing can commence. There is a need for an execution display after firing, this being important to confirm the correctness of the status at the marked places and whether transitions are enabled. This display should include current time stamps and be increased after each cycle.

6.0 ANALYSIS OF THE SHARED-TRANSITION MODEL

6.1 Types of Analysis

There are two main types of analysis carried out during Petri-net simulation, viz: (i) verification of the model correctness and (ii) performance evaluation, using times Petri-nets. The nature of the simulation enables the user to interact with the real time system during the simulation. the power of this concurrent modelling technique lies in its ability to detect deadlock conditions, conflicts and the boundedness of Petri-nets. Deadlocks occur when no transition can fire, resulting in the simulation being suspended.

Correctness of communication protocols can be established by observing execution of the simulation. Jobs or parts are normally sequenced through a series of workstations on a production line. Parts are transferred from one station to the next via independently controlled transfer devices, such as robot and conveyors.

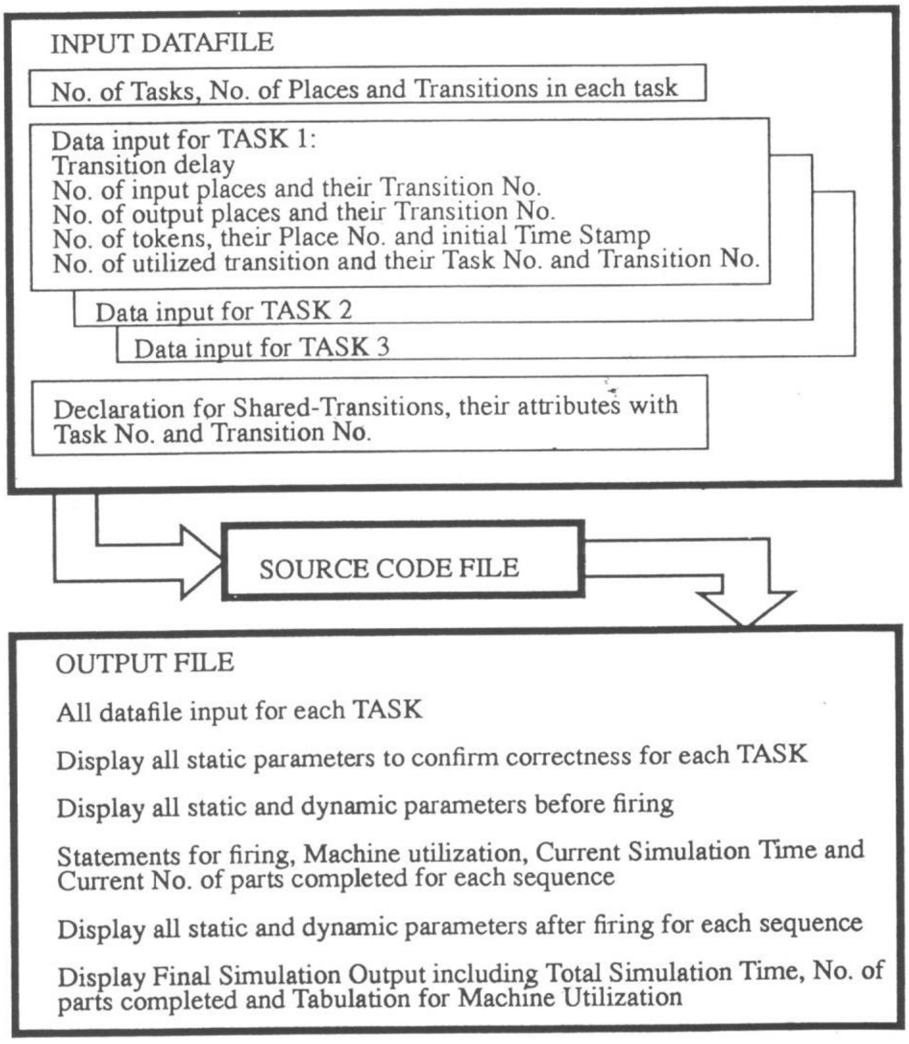
INPUT DATAFILE
No. of Tasks, No. of Places and Transitions in each task
Data input for TASK 1:
Transition delay
No. of input places and their Transition No.
No. of output places and their Transition No.
No. of tokens, their Place No. and initial Time Stamp
No. of utilized transition and their Task No. and Transition No.
Data input for TASK 2
Data input for TASK 3
Declaration for Shared-Transitions, their attributes with Task No. and Transition No.
SOURCE CODE FILE
OUTPUT FILE
All datafile input for each TASK
Display all static parameters to confirm correctness for each TASK
Display all static and dynamic parameters before firing
Statements for firing, Machine utilization, Current Simulation Time and Current No. of parts completed for each sequence
Display all static and dynamic parameters after firing for each sequence
Display Final Simulation Output including Total Simulation Time, No. of parts completed and Tabulation for Machine Utilization

Fig. 5 Data Structure and Programming Procedure

The structure of shared-transition Petri-nets, their expressiveness and their manner of processing, enables the implementation of a modular knowledge base of rules and the development of simulation tools for manufacturing systems can also be represented as a set of hierarchically organised subnets, where each subnets can be isolated and analysed independently. This capability offers flexibility in the sense that systems can be expanded independently and changes made to segments to the design hierarchy.

6.2 Execution of the Shared-Transition Model

Contributing input to each shared-transition will be single transitions shared by at least two tasks. The transition delay for these single transitions should be scheduled in the normal way. However the transition delay for a shared-transition (*ST*) will be the maximum value among the contributing single transitions. The sequence of for each contributing single transition to be enabled will also be scheduled in the normal way. An *ST* will only be enabled when all the incoming *T* are enabled. then all the time stamps for output places from this *ST* will be the addition of its transition delay (which is the maximum value among the contributing single transitions) with the maximum time stamp among all the input places to the contributing single transitions. furthermore the current time stamp attached to the output of an *ST* should be compared with the current time stamp of each single transition (*T*). The maximum time value of the two will become the current total simulation time for the modelled system. Hence, for example the number of parts completed could be calculated when the Petri-net schema has included the total flow of tokens from start to finish. During a certain simulation period, the utilization of each machine can also be analysed, by knowing the total transition delays related to each task.

Currently the modular Petri-nets simulation capability produced by the first author is written in the c programming language an executes sequentially. Using this capability the concurrent movement of tokens in each task can be examined clearly. However, further advantage would be enabled if the simulation programs developed could be executed concurrently and in real time. Current work is therefore considering the potential use of a multiprocessor execution environment, potentially with multiprocessor systems. However, the current *UMC* configuration editors do not support multiprocessor execution of tasks.

7.0 RESOURCE-SHARING MECHANISM

7.1 Current Resource Sharing Mechanism Adopted

A manufacturing machine system often contains machine systems which share resources (Zhou et al 1990). For example, in a production line, two machines may share a common robot for loading and unloading. When a system contains shared resources, inappropriate allocation of these resources may result in a system deadlock. Particularly when a system contains a sequentially shared resource, it may become blocked when too many parts are input during a specific time interval. In other words, the system behaviour may depend not only on its organisational structure but also on its initial status, i.e., initial token distribution in its Petri-nets equivalent model. Synchronization functions need to deal with potential conflicts (the case where several processes which share resources are enabled) by taking care that only one transition can occur out of those that wish to share a resource (Dotan 1991). to be closer to the actual way in which process dynamics occurs, it is assumed that several events that arise in machines systems can occur and execute in parallel, this implying that several transitions can be fired simultaneously as long as they do not share resources. Yim and Barta (1994) explained that the concept of an inhibitor arc could play a role in preventing the firing of any transition to prevent the transition from firing when the connected place has tokens. In a case where a number of transitions having share a resource, only one resource sharing transition should be enabled at a time to avoid conflict. Any transition firing could be based on the priority or sequence of the modelled system.

The approach used by Dotan (1991) for the modelling of resource-sharing machine systems is to first activate the transition having the shortest firing delay time. Furthermore, several transitions can be fired simultaneously of they are not in conflict and if they have the same firing time. If several enabled transitions share resources and have the same firing time, the one which requires the least number of resources will be fired. An enabled transition can move to either a fired state or disabled state. It will move to a fired state if it is not in conflict with another transition (i.e. they do not share resources). An enabled transition will move to a disabled state if there is another enabled transition sharing resources with it, but having the shorter firing time. After the enabling of all transitions has been checked in parallel, a clock is initiated and the transitions compete for firing. The transition with the shortest firing time will stop the clock upon being fired, disabling the rest of the enabled transitions. This approach seems exhibit excellent properties with respect to avoiding any conflict and deadlock. However, problems can occur in respect to job queues, especially where resource-sharing machines have to handle multiple types of jobs at the input and output of the complete modelled system. The sequence of flows for different types of token (parts) must be examined clearly. This sequence has to be scheduled and be incorporated correctly in a datafile before it can be executed. It is good to incorporate a minimum time for firing, this to enable a minimum time in real processes without any conflict and deadlock.

The flow of events in a real system is similar to that found in the computational processes which execute when running the simulation program. an event (transition) is enabled when the set of precondition (marked input places with token) is fulfilled. A schedule related to routing sequence determines the order of a queue when an event should be enabled. a set of sequence numbers will only allow one shared place transition to be enabled, this is to avoid any contradicting messages from other events sharing resources with it. A means of allowing a random choice of resource-sharing transitions that to be enabled could be adopted in an effort to enable only one transition to be fired a time. However the sequence number generated in a random way may not correspond to machine sequence which can be operated without deadlock. After an event is fired, the tokens of each associated precondition and post conditions should be increased by 1 or decreased by 1 respectively.

7.2 Programming Procedure Adopted for Resource Sharing

The programming procedure devised to handle the problem of resource sharing is as follows:

1. Open two files for inputting data and record all execution results. The input data for task no., places no. and transition no., which describe the Petri-net schema in each task of machine systems are displayed, including their attributes. For example, the attributes for a transition includes a time delay, whether it become an attribute to a shared-transition or not (i.e. shared status), and whether it can be

considered to calculate a machine utilization or not (i.e. utilize status. In the case of places their attribute includes time stamp, whether it is a shared resource or not, and marked tokens to indicate object movement. All these data can generally be considered to be static data. A dynamic data such as an enabled status for a transition and a marked token in a relevant place is displayed as a 1.

2. Prepare a read input file for transferring data into a source code file, which needs to be modular in structure in order to easily make changes.

3. Prepare a function for a Petri-net simulation with looping. Firstly before looping, execution all input data in an output display. This is important to check and confirm that all input data are correct. Within the loop, the following programming procedures are carried out.

   (a) All single transition are checked to see if all their input places are marked and indicate their status as 'ON'. Then check whether all resource sharing transitions are enabled and their sequence statements for firing are scheduled.

   (b) All shared transitions are checked to see if al their attributed single transitions are enabled or not, if enabled then return status as 'ON'.

   (c) Display static and dynamic data before firing. The dynamic data should include the enable status for transitions, the marked status and time stamp for all places.

   (d) Calculate the sum of the utilized transition delay in each task and if the relevant transitions are enabled. This is done to facilitate calculation of the machine utilization.

   (e) Prepare a function for firing a shared transitions (*ST*) for firing. In this function, firstly determine the longest delay among attribute shared transitions (incoming single transitions). Secondly, generate a new time stamp on the outgoing tokens according to the largest time stamp on the incoming tokens. Thirdly, display the firing statement for *ST* with its value of transition delay. finally, disable all *ST* attributes and unmarked all tokens at the input places. then mark the outgoing places to indicate objects movement and also disable the *ST*.

(f) Prepare a function for firing of single transition (*T*). All procedures are similar to the *ST* explained above, except a single transition use a determined transition delay earlier in the input data.

(g) Find the maximum current time stamp by comparing the *ST* output and T output. this is to record the current total simulation time of any current sequence.

(h) Analyse the utilization of each machine or task by dividing the current total transition delay by the current simulation time.

(i) Calculate the no. of parts completed by observing the presence of a token at the output place. This output place is an indication to determine the process of a job or part is completed. this is done by observing the total token flow from start to finish.

(j) Display the execution after firing.

4. Display the simulation output, indicating the total simulation time, no. of parts completed and list machine utilization values.

7.3 Simulation Algorithm and Programming : Example Use in a Machine Cell

The simulation algorithm used a handle conflict is either : identifies all the resource sharing places and their related transitions then fires only one transition at a time; or by checking for each task, identifying the number of resource sharing places fires only one transition connected to each resource sharing place. To illustrate this phenomena refer to Fig. 6. The resource sharing machine system is represented by the Petri-net analysed and shown in Fig. 7. Here the machine system is divided into five tasks. There are four places at which that are to be shared resources, namely P11, P21, P44 and P54. For example the shared place P11 shares two transitions, i.e. T11 and T12. However only one of these two transition will be enabled at a time and all transitions related to a particular resource-sharing will be scheduled in sequence. As seen from Fig. 7, the order of firing of the resource sharing transitions (i.e. ST1, ST2, ST4 and ST7) connected with the resource sharing place P21, must be determined prior to running the simulation. Here the aim will be to achieve the highest utilization of a machine. Figure 7 shows that Machine A and Machine B are accessed by the robot on an alternating basis in order to complete more parts in a given processing time. Figure 8(a) shows sample output from the simulation where Machine A and B utilisation data is obtained from an analysis carried out for the machine cell.

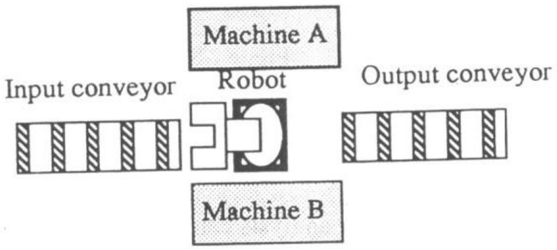

Fig. 6 A Shared Resource Machine Cell Layout

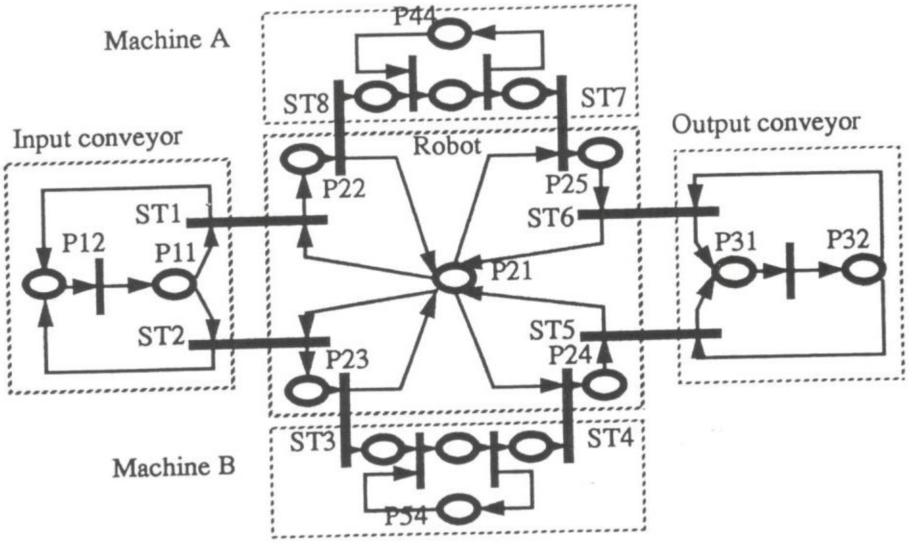

Fig. 7 Modular Petri-net Model of the Machine Cell

```
<< SIMULATION OUTPUT >>
Total simulation time = 38.0
No. of parts completed = 2

 TASK    M/C TME    % UTILIZE

 1       2          5.26
 2       38         100.00
 3       3          7.89
 4       5          13.16
 5       5          13.16
```

(a)

```
<< SIMULATION OUTPUT >>
Total simulation time = 58.0
No. of parts completed = 2

 TASK    M/C TME    % UTILIZE

 1       2          3.45
 2       48         82.76
 3       3          5.17
 4       10         17.24
```

(b)

Fig. 8 Simulation Outputs (a) with Machine A and Machine B (b) with Machine A Only

Further analysis can be carried out to see what happens if the machine cell in Fig. 6 is operated without Machine B. Here the simulation output (as shown in Fig. 8(b) indicates that, in order to complete two parts the total simulation time is higher (being 58 as compared to 38 under previous conditions), the robot is less utilized and Machine A has to work double time.

8.0 INTERPROCESS COMMUNICATIONS USING UMC

In a multi-tasking applications, processes must work together to perform overall job functions. This requires the passing of data and synchronization between the processes. A process (or task) has the job of forking the other process such as by using message passing mechanisms, which make up the application, and may themselves fork another task (Tsujino et al 1984). For example, it may be important that one process must not continue its job until another process must not continue its job until another process has collected data it requires. The use of such message passing mechanisms should be harmonized with that of the programming language used, in this case the 'C' programming language. the functions of synchronization and data transfer are known as inter-process communication (*IPC*). Correct use of these functions is essential to the operation of a multi-tasking application. OS-9 (Dayan 1993) has several different interprocess communication methods available for use by application programs. this *IPC* provides a send statement and a receive statement, which are basic primitive elements of a message passing mechanism.

By splitting the application tasks into separate programs, which execute as separate processes, another problem has been introduced. A process must be able to exchange data with other processes, and must be able to activate a sleeping processes, and must be able to activate a sleeping process when it has data ready for that process. In a send statement, a sending process must designate a receiving process by its process identifier. In a receive statement, a receiving process may or may not designate a sender process by its process identifier. This is the purpose of *IPC*. The *IPC* mechanisms include signals, events and data modules. It is possible to combine signals with user data modules.

UMC provides mechanisms for *IPC* which can be used by programmers: namely events, signals and user data modules. Events are operating systems efficient (Carrot et al 1993), and are well suited to situations where there are control flows between tasks. However, as with the underlying OS-9 events, the *UMC* events can only wait for one event at a time. It only a few events need to be waited for then it is possible to set up a hierarchy of events. However, this approach it not ideal as it involves polling of other events to see which one has changed value. Whilst waiting for a *UMC* events tasks are suspended until either the event wait condition is fulfilled or until a signal is received.

Signal provide another way of OS-9 and *UMC* controlling the execution of multiple asynchronous processes and protecting shared system resources from simultaneous access by several concurrent processes. A task may be signalled sequentially to multiple destinations and single task may receives signals from more than one process. *IPC* using signals in *UMC* is based around what is already provided by standard OS-9 functions. As with any OS-9 process which requires to receive signal, *UMC* application tasks must install a signal handler before receipt of any incoming signal, otherwise such a signal may cause the task to exit. A signal may be sent from one task to another task using the OS-9 *kill()* function. One of the parameters of this function is the process identifier (pid) of the receiving tasks, using *umc_link_task()* functions and reading its 'pid' from its task sub-structure in the machine data module, using *umc_rdmodi()*. This machine data module should be written by 'mc' when it forks the task.

9.0 SCHEDULING AND CONTROL OF MACHINE SYSTEMS

In this study the authors considered manufacturing machine problems from three related viewpoints, namely: planning, scheduling and real-time control. The question faced is how can this be accomplished in the best possible way. Planning issues concern long-term problems including the loading, grouping and selection of jobs in manufacturing machine systems (Basnet and Mize 1994). Scheduling problems may include resource allocation problems over a shorter time horizon. Real-time problems are concerned with intimate control of the real processes. Generally speaking modular, hierarchical control structures can be mapped appropriately onto the layout of a planned manufacturing systems and lead to clear and workable control solutions. Each modules of such a system and relevant events and states of each machine can be represented, activated and communicated by using Petri-net specification technique. The scheduling of part movement can be resolved by identifying the firing sequence or queue of tokens. The problems of mapping the Petri-net modelled system onto the real hardware and software systems can then be enabled using *UMC*, as developed by other *MSI* researchers at Loughborough University.

10.0 BENEFITS FROM USING THE SIMULATION TOOLS

The simulation tool was designed and developed in a modular and flexible form which can be used in both the design and operational phases of manufacturing systems. During design a system, wide description of events and states related to each machine is handled by a Petri-net model associated with each module or task. Communication between each module is then accomplished via shared events with tokens for jobs or parts transferred to other module. In a real system such tokens follow a sequence from start to finish. During implementation and operational phases, transition delays in the Petri-net models are directly translated into the motion of parts, this essentially being accomplished by the drive elements built within transportation machinery (such as robots, conveyors, etc.). In these phases, not only the current states of a machine system are mapped to the states of tokens in the Petri-net models, but each physical machine is mapped directly onto its own machine control module. Utilization of the *UMC* protocols and tools enhances further the flexibility offered during the design and implementation of machine control systems, this in a generalised and consistent manner. Thus, the modular Petri-net tool produced by the first author demonstrates the following benefits:

By closely mapping characteristics of a real machine system into a modular form of Petri-net model, not only is the modelling process more easily understood but also implementation of control logic is more readily accomplished by converting the machine control logic into a simulation model contained within each module. This simplifies the design process and helps to evaluate any problems which may arise, especially in regard to conflict and deadlock. Conflict is respect of parts movement can be resolved by shifting the position of individual jobs in the scheduling sequence. Thus, the simulation model helps to more definitively design the orientation of a real production system.

By separating processes of the real systems into modular elements and models improved flexibility in terms of defining new hardware components can be facilitates, this by adding a new modules during the modelling process. New data for each new machine is also easily added into the relevant input data file without the need to change other data previously input. Furthermore no changes are required to the source code which will drive the simulation.

By having modular and flexible machine control logic as part of the proposed simulation model, a real machine system can be scheduled and reconfigured rapidly to produce multiple part types within transfer lines. For example of an input conveyor is be added into a manufacturing cell in which it is necessary to process a different type of part, the modelling process can be easily carried out by mapping this additional machine into a corresponding modular Petri-net. Communication between machine components can then be handled by shared-transition nets. Thus the simulation tool enables a large number of multiple part types to be analysed within a given simulation period.

A fundamental requirement of flexible manufacturing machines is to achieve appropriate levels of flexibility whilst manufacturing certain type of parts with acceptable short times. Here processing time are highly dependent on the machine rate. Thus, when designing a manufacturing machines it is considered an important criteria to complete a certain product at the shortest time possible. The simulation tool offers a simple way of experimenting with and analysing resultant transition delay in the simulation model. This machine data input may be collected from the actual manufacturing machines, especially in terms of their motion rates. Thus machine performance criteria and utilization rates can be optimised. This form of simulation study can play an important role in maximising the productivity of a manufacturing process. This enabling a choice of the most suitable machines available in the market to be made before purchasing such equipment or helping design the machine in cases where special purpose machine system needs to be engineered.

11.0 CONCLUSION

A shared-transition Petri net based simulation tools is described which can be used to integrate and evaluate machine control systems. In this simulation tool, a given hardware component layout can be mapped similarly onto Petri-net objects within a modular structure. The shared-transitions clearly identify the interfaces and communication paths between resources used in real systems. mechanisms are included for handling resource sharing. Furthermore, the modularity of the Petri-net modelling tool with shared-transitions facilitates the addition or deletion of machine hardware. The data input to modelled system can also be supplied in modular form, this making changes and analysis much easier during planning and operational phases.

REFERENCES

1. Albus, J.S., McGain, H. G., Lumia, R., NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM), Nat. Inst. Standard and Tech, Tech Rep 1235, Gaintherburg MD, 1989.

2. Albus, J. S., Outline Theory of Inteligence, IEEE Transaction on Systems, Man and Cybernetics, 21, 473-509, 1991

3. Ariffin, S., Weston, R. H., Harrison, R., A Task Level Architecture for Machine Control Systems, Procs of the IASTED International Conference Applied Modelling and Simulation, Lugano Switzerland, pp 204-207

4. Ariffin, S., Weston, R. H., Harrison, R., A Modular Petri-net Approach to the Design of Distributed Machine Control Systems, Procs of the 31st MATADOR Conf., UWIST, UK, pp 621-626

5. Basnet, C., Mize, J. H., Scheduling and control of flexible Manufacturing Systems: A Critical Review, Int. J. Computer Integrated Manufacturing, 9, 340-355, 1990.

6. Carrot, A. J., Morley, T., Booth, A. H., UMC 3.0.3. User Guide Rev. 5, (MSI Research Institute, Loughborough University of Technology, U.K.), 1993.

7. Dayan, P. S., The OS-9 Guru (A Galactic Production), 1993.

8. Dotan, T., Ben-Arieh, D., Modelling Flexible Manufacturing Systems: The Concurrent Logic Programming Approach, IEEE Transactions on Robotics and Automation, Vol. 7, No. 1, 135-149, 1991.

9. Dotan, Y., Using Flat Concurrent Prolog in System Modelling, IEEE Transactions on Software Engineering, 17, 493-512, 1991.

10. Johnson, T. L., Baker, A.D., Trends in Shop Floor Control: Modularity, Hierarchy and Decentralization, Procs 5th IEEE Int. Symposium of Intelligent Control, 2, pp 45-51, 1990.

11. Krogh, B.H., Genter, W.L., Petri Net Analysis of a Transfer-Line Protocol, Renssealaer's Second Int. Conf. on Computer Integrated Manufacturing, New York, pp 189-192, 1990.

12. Tsujino, Y., Ando, M., Araki, T., Tokura, N., Concurrent C: A Programming Language for Distributed Microprocessor Systems, Software-Practice and Experience, 14, 1061-1978, 1984.

13. Wiendahl, H. P., Garlichs, R., Zengragen, K., Modelling & Simulation of Assembly System, Annals of CIRP, 40, 557-585, 1991.

14. Williamson, R., Horowitz, E., Concurrent Communication and Synchronization Mechanisms, Software: Practice & Experience, 14, 135-151, 1984.

15. Yim, D.S., Barta, T.A., A Petri Net - Based Simulation Tool for the Design and Analysis of FMS, Journal of Manufacturing System, 13, 251-261, 1994.

16. Zhou, M.C., Dicesore, F., Rudolp, D., Control of flexible Manufacturing System using Petri-nets, IFAC Symp. Series - Procs. of a Triennial World, pp 47-52, 1990.

17. Zhou, M. C., Dicesore, F., Guo, D., Modelling and Performance An Analysis of a Resource-Sharing Manufacturing System using Stochastic Petri Nets, 5th IEEE Int. Symp. on Int. Conf., Philly, Pens., 2, pp 1005-1010, 1990.