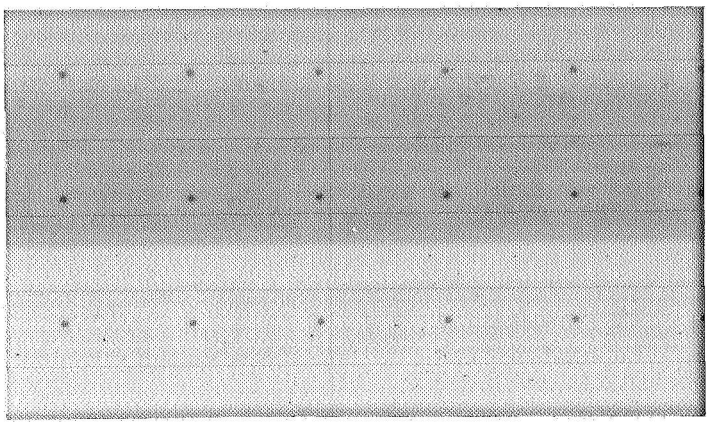
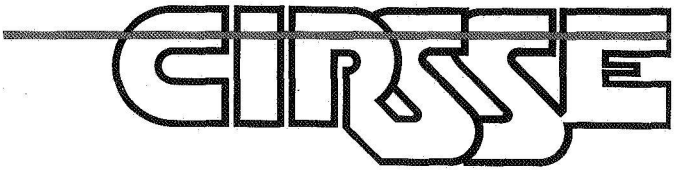
NASA-CR-193196





Center for Intelligent Robotic Systems for Space Exploration

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

PETRI NET CONTROLLERS FOR DISTRIBUTED ROBOTIC SYSTEMS

by

D. R. Lefebvre and George N. Saridis

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering Department
Troy, New York 12180-3590

September 1992

CIRSSE REPORT #126

Petri Net Controllers for Distributed Robotic Systems

D.R. Lefebvre G.N. Saridis

Electrical, Computer, and Systems Engineering Department
Center for Intelligent Robotic Systems for Space Exploration
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Abstract

Petri nets are a well established modelling technique for analyzing parallel systems. When coupled with an event-driven operating system, Petri nets can provide an effective means for integrating and controlling the functions of distributed robotic applications. Recent work has shown that Petri net graphs can also serve as remarkably intuitive operator interfaces.

In this paper the advantages for using Petri nets as high-level controllers to coordinate robotic functions are outlined; the considerations for designing Petri net controllers are discussed; and simple Petri net structures for implementing an interface for operator supervision are presented. A detailed example is presented which illustrates these concepts for a sensor-based assembly application.

1 Introduction

The *Theory of Intelligent Machines*, as proposed in [8], describes a hierarchical organization of the functions of an autonomous robot into three levels: an *Execution Level* containing the hardware and basic control functions, a *Coordination Level* integrating the capabilities of the intelligent machine (IM) across hardware systems, and an *Organization Level* providing higher-level planning and reasoning capabilities. Figure 1 illustrates this hierarchical organization. An analytic formulation of the theory using information-theoretic measures of uncertainty for each level of the intelligent machine has been developed in recent years [9][10]. A long-term research goal of the Center for Intelligent Robotic Systems for Space Exploration (CIRSSE) is to attempt, guided by these theories, implementations of autonomous robots.

In the past year at CIRSSE a computer architecture was developed that implements the lower two levels of the intelligent machine [4]. The architecture supports an event-driven programming paradigm that is independent of the underlying computer architecture and operating system. This approach has enabled us to build heterogenous, distributed applications that integrate Execution-level and Coordination-level functions.

Execution-Level controllers for motion and vision systems were created which interface to testbed hardware comprising two 9-DOF arms (controlled as an 18-DOF manipulator) and 5 cameras. The control systems are highly modular and configurable in order to support a flexible program of research. Additionally, by basing both motion and vision control systems on the same architecture, integration among Execution-level functions and with upper level operations becomes feasible.

Robotic systems may be designed for a spectrum of modes of operator interaction ranging from simple teleoperation (as initially applied to nuclear fuel processing) to virtually complete autonomy (as exempilfied by NASA space probes such as Voyager or Galileo) with varying levels of intelligence. Intermediate in this spectrum is *supervised autonomy* – intermediate both in the degree of operator interaction and in the intelligence of the robotic system. When the requirements for this range of robotic applications are mapped onto the hierarchical organization proposed by the Theory of Intelligent Machines, we find that the Execution Level supports teleoperation, the Coordination Level is additionally needed for supervised autonomy, and the full hierarchy including the Organization Level must be in place to achieve autonomy.

While we are a number of years away from even modestly-intelligent autonomous machines, the foundation exists today to research operator interaction with the Coordination Level to effect supervised autonomy. Key research issues include the "division of labor" between machine and operator, the effect of communication delays for remotely supervised machines, computer and control architectures for distributed applications, mechanisms for improving reliability and robustness of robot functions, and the design of the interface with the operator [6][11].

The paper is organized as follows. This section has briefly described the hierarchical model of an intelligent machine that serves as an organizational design for this work. A Petri net implementation of the coordination-level of the hierarchical model is presented in Section 2. Section 3 covers recent work at CIRSSE aimed at integrating robotic functions through the use of Petri nets. Section 4 outlines Petri net structures that enable an operator to monitor and to interact with the robotic system. And, Section 5 presents an extended example based on an assembly application developed at CIRSSE.

2 Coordination-Level Petri Nets

At CIRSSE we address supervised autonomy research issues in the context of a hierarchically-controlled, but parallel, distributed robotic system. As discussed above, we have developed a computer architecture and Execution-level control systems that enable experimentation with an integrated robotic testbed. Earlier theoretical work [12][13] took a linguistic schemata approach to describe Coordination Level function, and used Petri nets to model its operation. Hence, our first attempts to implement the Coordination Level have also been based on Petri nets.

2.1 What is a Petri Net?

Petri nets (PN) are graph-theoretic tools for modeling the dynamic behavior of discrete event systems. Ordinary PN are directed graphs with two types of nodes called *places* and *transitions*, which are connected by *arcs* [7]. In a PN graph, places are represented by circles, and transitions appear as bars. A simple PN graph is shown in Figure 2.

Places may contain *tokens* that indicate the state of the PN. Tokens are moved between places by the firing of a transition. The PN software developed at CIRSSE employs a "two-phase" transition firing rule: 1) transitions are enabled when there are tokens in all input places of the transition, these tokens are removed when the transition fires; 2) firing the transition causes an action, that may result in a time delay; and 3) once the action is complete, token are added to the transition's output places. PN such as these which contain transitions that may require a probabilistic amount of time to fire are called Generalized Stochastic Petri Nets (GSPN).

A simple example will illustrate the operation of a GSPN. In Figure 2a there are tokens in places p_1 and p_2, the input places of transition t_1; hence, t_1 is enabled. When t_1 fires, the tokens are removed from p_1 and p_2, and the action associated with t_1 is invoked. CIRSSE PN displays indicate a transition "firing in progress" by inverting the transition symbol as shown in Figure 2b. When the action is completed, tokens are added to p_3 and p_4, the output places of t_1; resulting in the marking shown in Figure 2c. Now, transitions t_2 and t_3 are enabled and can fire in parallel. Transition t_4 will not be enabled until the actions associated with both t_2 and t_3 have finished and places p_5 and p_6 are marked. This example shows how PNs may be used to control parallel operations and enforce the precedence of operations.

Initial CIRSSE research was based on conventional GSPNs; however, all current research uses enhanced GSPNs that include an additional condition on the enabling of transitions. This added condition is implemented as an *enabling function* for each transition which is called whenever all of the transition's input places are marked and the transition would otherwise be enabled. The enabling function may check system status or test a loop variable and then return 'True' or 'False' to indicate whether the transition should fire. With enabling functions a GSPN can model deterministic and conditional programming loops, and can implement a Petri Net Transducer (PNT) [12].

Often enhancement to conventional theory results in a loss of analytical tools; such is the case with adding the enabling function to GSPNs. Fortunately, recent work is begining to close the gap [2], and work continues. Future PN enhancements may enable the construction of PN controllers with more automated error recovery, and of coordinator PN that incorporate feedback and learning.

2.2 Why use Petri Nets?

The original rationale for using Petri nets was based on their representational capabilities and the availability of PN analytical techniques. Later, as we gained experience using them, we found that Petri net graphs can serve as powerful and intuitive user interfaces. The primary benefits of PNs for integrating robotic functions and supporting operator supervision are:

- Petri nets can represent the parallelism of a distributed robotic system, both for modelling and for control. When coupled with an event-driven, real-time operating system, a PN can provide the basis for control software that fully exploits the capability of a distributed system to perform concurrent operations.
- Analytical techniques can determine the reversibility, liveness, and boundedness of the PN controller. These important properties respectively ensure that the system can recover to its initial state, is free of dead-lock, and correctly handles its resources [5].
- PN controllers can be built hierarchically from simplier subsystems while retaining their representational power and desirable properties. Of equal importance, complex PN may be abstracted as *reduced nets*, under reasonable restrictions [3].
- The state of the system can be ascertained directly from the marking of the PN graph. Further, the preconditions for an operation and the precedence of operations are readily apparent in the PN graph. Through judicious design, a reduced PN can be obtained that accurately represents the system in an intuitive form suitable for an user interface.
- PN display software can be built to allow the user to interact with the system to control the pace and sequence of operations. Later in this paper we will discuss simple PN structures that implement operator authorization/confirmation, and that enable the operator to assume manual control.

2.3 Dispatcher–Coordinator Structure

The role of the Coordination Level in the Theory of Intelligent Machines is to interpret and manage the plans from the Organization Level, add real-time details as appropriate, and communicate instructions to the Execution Level in order to coordinate the operation of the IM. In addition, the Coordination Level monitors machine performance to provide feedback for refining short-term decision making and to assist replanning when the nominal task plan is not successful.

As shown in Figure 1, the Coordination Level of the IM is organized into a tree structure, with a *Dispatcher* at the root and multiple *Coordinators* at the leaves. In

a fully autonomous IM, decisions are made at the Organization Level and are communicated to the Coordination Level through the Dispatcher. In a supervised autonomy application the operator directly interacts with the Dispatcher. In both instances, the Coordinators interface with their corresponding Execution-level functions.

Viewed from the bottom up, the Dispatcher is the first component of the IM that deals with the machine as a whole; and thus can coordinate actions across Execution Level functions, e.g. move a manipulator to a position determined by visual sensing. As its name suggests, the Dispatcher's primary function is to receive commands from the Organization Level or an operator, and to dispatch these commands to the appropriate Coordinator for implementation. Additionally, the Dispatcher must decompose high-level commands into a sequence of (often parallel) Coordinator operations, and to instantiate real-time values to the abstract values used in planning or that were input symbolically. This description is a simplification, however, as the Dispatcher is called upon to perform a number of other functions such as supporting system-wide communications, scheduling use of resources, detecting and correcting intermediate-level errors, and performing some run-time planning.

A Coordinator can be considered as an expert in applying the deterministic functions of a narrow domain of the Execution Level. For instance, the *Motion Coordinator* may have several strategies for moving the tool frame of a manipulator into a requested position and orientation (e.g. employing redundant degrees of freedom), and can choose the strategy with the highest probability of success based on current constraints of the environment (e.g. obstacles) and time goals. Coordinator operations correspond to underlying capabilities of the Execution-level hardware; and so do not change when the system is reprogrammed for a different application.

The Coordinators also play a role in error handling. While a nominal plan may be recommended by the organization-level planner, it is the Coordinator's responsibility to reliably accomplish the requested task in real time. Errors are handled first by the Coordinator, and are passed up to the Dispatcher only when a local strategy is not adequate to resolve the condition. In some instances the Dispatcher must turn over error resolution to the Organization Level where operations may be replanned; or, in a supervised application, to turn control over to the operator. Feedback is also provided through the Coordinators in order to update knowledge bases and improve decision making of the IM [10].

In building a PN implementation of the Coordination Level for a specific application, only the Dispatcher Petri net need be designed since the Coordinator PNs are reused. PN transitions of the Dispatcher represent actions performed by the Coordinators (such as move_robot or capture_image); while places represent states of the system (such as robot_at_ready_position or required_data_exists) [1]. The Dispatcher is connected to the Coordinator PNs through a *Coordination Structure* as discussed in [13]. The Coordination Structure consists of a set of transitions and places that control access to Coordinator resources and ensure that the thread of execution returns to the appropriate Dispatcher transition.

Transitions in the (predefined) Coordinator PNs represent actions performed by Execution Level control operations. When one of these transitions fires, a robot control function is called. The structure of the Coordinator PN ensures that the preconditions for a requested operation are met and run-time data values are supplied before the control action is performed. The next section will address how the PNs are integrated with the controllers in a distributed robotic application.

3 Integration of Robotic Functions

When the tasks of the hierarchical model of the intelligent machine (Figure 1) are assigned to the components of the testbed hardware, a superficially similar organization is obtained as shown in Figure 3. In reality, the IM hardware implementation has three characteristics that require a more complex organization that suggested by the hierarchical model:

- The system is *distributed* — comprising multiple, loosely-coupled processors, possible widely-separated spatially. For instance, part of the system may be in orbit while the remainder is ground-based.
- The system is *heterogeneous* — employing diverse processor architectures, operating systems, and communication media and protocols.
- The system spans a *wide range of time scales* — from synchronous sub-5 ms. control loops to aperiodic run-time planning and human time-scale interactions.

We suggest that these characteristics will be common, to varying degree, among most autonomous or telesupervised robotic applications.

This characterization describes the CIRSSE testbed well. Computing facilities in CIRSSE's robotics lab include two multi-microprocessor VME-backplane computers, and several UNIX workstations (Sun SparcStations and Sun 4s). The VME cages are used for real-time applications such as control of robot manipulators and camera subsystems, and for image processing. These real-time applications are implemented in the VxWorks operating system. UNIX workstations support user interfaces and higher-level IM functions such as path planning and the Coordination Level Petri nets. The VME cages and workstations are interconnected by a local network operating the standard ethernet protocol.

One of the challenges we faced in building the CIRSSE testbed was to integrate the heterogeneous computing environment. We sought a means to build distributed applications that span all of the laboratory's equipment and take advantage of the parallel organization of the subsystems.

3.1 Distributed Applications

To cope with the need to build distributed applications, we developed extensions to UNIX and VxWorks that effect a communication layer between the operating system and the application. A simple, uniform

programming interface was produced which encourages the development of modular functions that can be interchanged (to construct applications from standard components) and that can be moved among processors (to balance loads). These operating system extensions were named the *CIRSSE Testbed Operating System (CTOS)*.

In broad terms, CTOS supports three objectives: 1) distribution of tasks, 2) communication between tasks, and 3) synchronization of tasks and the application. An event-driven environment was implemented in which tasks execute independently and in parallel, and communicate by exchanging messages. CTOS provides five services: 1) a *Bootstrap service* loads and initializes software modules on each processor as specified via configuration files; 2) a *Task identification service* assigns a unique identification to every task and associates it with the task's symbolic name for later retrieval; 3) a *Message passing service* acts as the primary communication mechanism between tasks; 4) a *Synchronization service* provides high frequency, low latency execution of real-time software functions on VME cages; and 5) an *Inter-processor blocking service* constitutes a second faster-but-limited communication mechanism between processors on a VME cage based on a "distributed semaphore."

Control software for CIRSSE's two 9-DOF robot arms and for a 5-camera vision subsystem was written based on CTOS [4]. Software for executing Petri nets were also implementation with CTOS so that Coordination Level PN could be fully integrated with the Execution-level controllers.

The diagram in Figure 4 describes the integration of PN and controllers. When the `move_robot` transition fires, the PN task sends a CTOS message to another task, labeled the *service task*, instructing it to perform the move operation. The service task prepares and sends a message to the *robot controller* task; this message contains the run-time data specifying motion parameters, e.g. destination and speed. After the robot controller completes the requested operation it sends a reply message to the service task, which relays the reply to the PN task to acknowledge completion. Firing of the `move_robot` transition then finishs by adding a token to place `done`.

In this example, tasks are assigned to specific processors in order to access local capabilities, e.g. PN displays need X-window support and controllers need real-time connection to hardware interfaces, or to optimize performance by reducing data transfer volume. Many tasks, such as the service task, need only a general compute resource and can be assigned to either a VME cage or an UNIX workstation. In fact, a single version of source code can be written that may run on either type of computer. CTOS routes messages between tasks based on symbolic names given to the task by the user — the user need not know the destination task's location. Thus, the distributed application can easily be reconfigured and tasks moved from one processor to another by modifying a configuration file and restarting the application.

3.2 Data Object Manager

Figure 4 also identifies another task called the *Data Object Manager* (DOM). The purpose of the DOM is to store data generated in one transition for use by a later transition. In this example, an on-line path planner retrieves the robot's current position and goal position from the DOM, calculates a path specification, and then saves the path in the DOM. The existance of a valid path stored in the DOM is indicated by marking the `path_valid` place in the Coordinator PN. The `path_valid` place is an input place to a subsequent `move_robot` transition indicating that a valid path is a precondition for the move operation. Note that the `goal_pos` data object shown in this example was input to the DOM via an earlier transition operation not shown in the figure. The `goal_pos` can be defined through a sensor operation or input by an operator.

A token present in a "data place" indicates the existance of a valid data object in the DOM. A data object is created as the result of a transition action such as a path planning calculation or sensing operation or entry of information by an operator. Using the CIRSSE PN controller interface an operator may add and subtract tokens from places through X-window mouse operations. Data places are protected so that the operator may remove a token but cannot add a token, i.e. cannot create a data object "out of thin air." Data places are displayed as double-line or heavy circles in PN controller displays.

The DOM is designed as a distributed service with components executing on all UNIX and VME chassis of the application. By distributing the operation of the DOM the volume of data transferred can be minimized as DOM services can be performed locally when data source and consumer are on the same chassis. The DOM is not a database manager since it does not support search operations or store data in normal form. In fact, the DOM simply stores blocks of bytes and retains no information as to its content. By using PN places, as described above, we can ensure data integrity and obtain data locking features similar to those of a database manager.

3.3 Building Applications

Tools to assist the construction of PN controllers are currently being developed at CIRSSE. Since the Coordinators have a fixed PN structure representing the capabilities of the underlying Execution Level system, the Dispatcher PN can be built from standardized components. It is feasible to develop a menu-driven tool that would enable the user to interactively generate the Petri net. This possibility is suggested by Figure 5.

In this example the user is specifying a control sequence to move the robot to a position determined by visual sensing. In Figure 5a the PN structure for the LOCATE operation is generated by choosing Locate/Arm_camera from the menu. The PN structure for the MOVE operation is added in Figure 5b. And finally, the user equates pairs of places to joint the LOCATE and MOVE operations to form the PN of Figure 5c.

The PN structure in Figure 5 is the reduced PN used for the controller display; the underlying PN

is much more complex. The tools for constructing PN controllers must also build connections between Dispatcher and Coordinators, implement calls to the DOM, and add structure for operator interaction (as will be discussed in Section 4.2). Eventually, the PN building tools will allow interactive development whereby the controller is run, modified, and rerun on a live system without rebooting between trials.

3.4 Recent Results

Two integrated applications have been built to demonstrate the capabilities of this approach. A camera calibration application was developed in which a light source grasped by a 9-DOF robot arm is moved within the work space of a stereo pair of ceiling-mounted cameras, and positional data is collected for later calculation of camera calibration parameters. This application requires cooperation of both the manipulator and vision subsystems (e.g. the robot must stop while the cameras capture an image), and can take advantage of some parallelism (e.g. querying the robot joint position while finding the light source in the image). This application was intended to demonstrate the basic integration concepts, and had little operator interaction.

The second application is a representative subsequence of an assembly task in which a strut is to be inserted into a partially completed strut-and-node structure. (This task was originally selected as relevant to space-based construction, however revisions to Space Station Freedom have deemphasized strut-based designs.) The application requires a high degree of integration because the locations of parts and the structure are determined via the vision system rather than by using taught points or CAD models. The task also offers an opportunity to investigate operator interaction as the system may be run autonomously or "single-stepped" allowing the operator to judge successful completion of a subtask and to intervene if necessary. This application will be discussed in detail in Section 5.

The Coordination Level PN for first application contains 50 places and 26 transitions. In the second application there are 168 places and 103 transitions. The PN for both applications were constructed manually; and the limits of the manual tools were reached in the second case. Our experience indicates that more versatile applications will require approximately an order of magnitude larger number of places and transitions, i.e. several thousand of each. To be practical, the synthesis of these large PN must be based on standardized subnets, and their construction must be automated.

4 Operator Supervision

In this section we present the PN structures that can be added to Coordination Level PN controllers to implement an interface for operator supervision. Three issues are addressed: monitoring system state, authorizing or confirming a machine operation, and detecting errors and recovering via manual intervention by the operator.

4.1 Monitoring System State

System state may be read directly from the PN graph. The presence of a token in a PN place asserts the condition represented by that place. For example, a place can represent a resource such as the availability of a robot arm. When the place is marked (i.e. contains a token) the robot is available; and if the place is empty then the robot may not be used. Our analysis has found that places can be categorized into four classes: places representing existance or validity states of *resources* and *data objects*, places corresponding to *control* states such as "operation done," and places symbolizing *operator input* states of authorization or request for manual control.

System state is changed by firing a transition. For instance, a transition may be assigned to "reserve the robot," then when the transition is fired a token is removed from the "robot available" place and added to the "robot in use" place; thus indicating the change of system state. In the case where a transition can not complete firing instantaneously, the transition symbol may be highlighted as seen earlier in Figure 2b. Hence, a highlighted transition indicates a system state of a particular "operation in progress."

Coordination Level PN controllers can be highly complex Petri nets with a large number of places and transitions. For this reason, the operator will usually interact with a reduced net representation of the PN controller. In a reduced net a number of places and transitions are abstracted into a simpler graph, perhaps as simple as a single transition. In this way the state space appears to be reduced for the purpose of simplifying the user interface. At times the operator will want to access the underlying complex PN; and so the interface software should support expansion of the reduced net to the full PN graph.

Another valuable feature inherent to the PN operator interface is that the precedence of operations is also apparent in the directed graph of the PN controller. The preconditions required to perform an operation are represented by the input places to that operation's transition, and the sequence of operations needed to satisfy the preconditions can be read directly from the PN. Alternative operation sequences for achieving a desired state (e.g. via manual or automated processes) can be built into the PN controller giving the operator a choice.

4.2 Authorization/Confirmation

Input places can be added to transitions to provide a means for an operator to authorize that an operation proceed, or to confirm that an operation was successful or not, and to choose the appropriate subsequent sequence. Further, the interface can allow the operator to add and subtract tokens from the places of the PN. In our UNIX X-windows implementation the operator moves the mouse cursor over a place and changes the number of token via the mouse buttons. A touch-screen would be an intuitive interface for this application.

To illustrate this construct we will consider the example of a simple move operation. The reduced MOVE subnet in Figure 5b expands to the portion of the PN controller shown in Figure 6. The operation begins when the `request move` and `Position` places are marked. Recall that a token in the `Position` place indicates that a previous operation created a descrip-

tion of the goal position for the robot movement and stored the position in the Data Object Manager. The `request move` place could have been marked by the operator or by the PN controller as the output of an earlier transition. The marking of `request move` and `Position` places enables the `PLAN PATH` transition which fires, plans a plan, saves the path in the DOM, and then marks its output places `request move` and `Path`. (Note that if a preplanned path had been available then the PLAN PATH operation could have been bypassed by marking the `Path` place rather than the `Position` place at the start.) At this point the `MOVE` transition is enabled *if the* `authorize` *place is marked.* The operator can "preauthorize" the move operation by injecting a token into the `authorize` place ahead of time as shown in the figure; or can specify that the system wait for operator authorization by leaving the `authorize` place empty. Once the `MOVE` transition has completed, the `wait` place will be marked; and the move operation can be finished by adding a token to the `confirm` place, thus allowing the `FINISH` transition to fire.

An alternative mechanism for providing operator control is to execute the PN controller in *single-step mode.* When the PN controller is in single-step mode, enabled transitions do not fire automatically. The operator must point to the enabled transition with the mouse cursor and then click the mouse button to cause the transition's firing. (This too is a good touch-screen application.) The advantage of this mechanism is that input places need not be added to every transition for which operator authorization or confirmation is desired. The disadvantage is that the entire PN must be in single-step mode, and so the operator must laboriously authorize each and every step. To alleviate this disadvantage we partition the PN controller into subnets that execute continuously and separate subnets that are set to single-step mode. Typically, the Dispatcher PN runs single-step and Coordinator PNs are put into continuous mode. For maximal flexibility, both operator interaction mechanisms are used; specifically, input places for operator authorization are included but are initialized to be nonempty, and the operator can switch between single-step and continuous modes at any time for any controller subnet.

4.3 Error Detection/Recovery

A long-term goal of CIRSSE research is to add more automated error detection and recovery to all levels of the intelligent machine with the aim of achieving more robust operation. At present it is practical to contruct Coordination Level PN controllers to enable an operator to perform error detection and recovery.

One means of doing this is suggested by Figure 6 where a `TELEOPERATE` transition has been included. Following completion of the `MOVE` transition, the operator has the choice of marking the `confirm` place to instruct the system to continue and finish the move operation, or mark the `manual` place to invoke teleoperation. Conceptually, the operator would inspect the result of the robot movement and then decide whether to accept it or to correct it manually. The PN controller is designed such that the state of the PN is identical following teleoperation as it would have been if the operator had not intervened.

It may not be possible (or at least highly impractical) to anticipate all failure modes of a system. Therefore it is important that the controller can be restored to match the system state. With a PN controller the operator can halt automated execution at any time, perform a manual operation or repeat a subsequence of the PN, and then adjust the marking of places to reconcile the PN controller with the actual system state. This ability to intervene and repeat a subsequence of operations has proven very useful during development and debugging of applications.

5 Example Application

An example will illustrate the use of Petri net controllers to coordinate a sensor-based assembly application. CIRSSE's testbed hardware consists of two 9-DOF robots, plus cameras mounted on the ceiling and on robot wrists. Only one of the two arms is used in this application. Independent control systems are implemented on separate VME cages for robot and vision subsystems as described in [4]. Petri net controllers executing on Sun workstations serve to coordinate the operation of the independent subsystems. The resulting control architecture is shown in Figure 3. A diagram of CIRSSE's testbed hardware appears in Figure 7.

The application starts with a partially-completed strut and node assembly including two legs of an equilateral triangle. The objective is to add the last strut to complete the triangle. Locations of all struts and nodes are determined via the vision subsystem. The sequence of operations is:

1. Initialize independent subsystems for camera, robot, and path planning.
2. Use ceiling-mounted stereo cameras to find location of partially-completed strut assembly.
3. Use robot wrist camera to refine positions of nodes which will receive the next strut.
4. Locate and pickup strut from rack.
5. Move to assembly and insert strut.
6. Return robot to initial position and shutdown all subsystems.

The overall application (excluding initialization and shutdown) represents an elemental operation for the construction of larger strut and node assemblies. With Petri net controllers we can abstract these operations as a single step within a larger sequence of assembly operations.

Application-wide coordination is provided by four PN controllers: dispatcher, motion coordinator, vision coordinator, and path planning coordinator. Figure 8 shows the *Dispatcher* PN controller display which serves as the user interface for this application. The display contains 47 places and 28 transitions which abstract the PN controller which actually contains 118 places and 71 transitions. Even with this level of abstraction this display is (intentionally) somewhat more complex than needed so that some underlying features may be illustrated. The three coordinator PN controllers, which service requests from the dispatcher, will not be discussed in this example.

The dispatcher PN has been folded into a reverse "S" shape to save space; thus, the sequence of operations is read from upper left to lower right. Operation of the three independent subsystems (camera, robot, and path planner) are represented as parallel paths which intersect when synchronization is required. The application starts when the **req_demo** place is marked by clicking the mouse button while the mouse pointer is over the place.

Firing the **start** transition enables the parallel initialization of camera and robot subsystems (transitions **init_CAM** and **init_ROB**). Initialization of path planner (**init_PP**) waits until the robot is initialized so that the planner world model can obtain current robot position. At this point the **CAM_ready** and **PP_ready** places are marked indicating completion of their initialization. The final initialization step is to ensure that the robot is in its home position. While the marking of **PP_ready** place enables both the **homed?** and **not_homed?** transitions, their enabling functions (Section 2.1) ensure that only one will fire. If the robot is not currently in home position then **not_homed?** and **move_home** will fire thereby moving the robot home.

The sequence continues with firing of transition **find_V** which requests that the vision subsystem use the ceiling-mounted stereo cameras to find the location of the partially-completed triangular strut assembly. Successful completion of this operation produces data objects for the 3-D position and orientation of the assembly nodes that will receive the next strut. Places **node1_pose** and **node2_pose** are marked when these data objects are stored in the Data Object Manager.

Across the middle of the PN display, read right-to-left, a sequence of operations calculates a view approach position (**calc_appr**), plans a path to this position (**PP_node**), moves there (**move_node**), and then refines the measurement of node position using the robot wrist camera (**refine_node**) for one node and then the other. The calculation of view approach position uses the estimated node pose obtained from the ceiling cameras. The node pose data object is replaced by an updated estimate following transition **refine_node**. This sequence of operations has two parallel paths: one for camera and robot, and the other for path planning. Camera and robot operations must be synchronized because the robot must be at viewing position and stopped before the wrist camera can capture an image. Planning of the path between node 1 and node 2 can proceed immediately after the path to node 1 is planned, however, because the start and end positions of the path are known, i.e. the two node poses found by the ceiling camera.

At this point let us make a few observations. First, since ceiling and wrist cameras produce essentially the same information, albeit at different resolutions, we should be able to bypass all operations between **V_found** and **node2_done**. This may be accomplished by switching the dispatcher PN controller to single-step mode during **find_V**, deleting the token from **V_found**, and then adding tokens to places **node2_done** and **PP_rdy3**. We have demonstrated this alternate execution sequence, but have rarely obtained successful strut insertions due to the poorer resolution of the ceiling cameras. Second, for an actual application we probably don't need to see all intermediate steps leading to the update of node pose. We can easily abstract these into a one transition of the dispatcher display for each node. This use of abstraction is illustrated by the **move_rack** transition which represents a sequence of operations that use the wrist camera to find a strut in the strut rack and move the robot into position to pick it up.

In the final sequence, shown across the bottom of the dispatcher display, the robot is instructed to pick up the strut (**pickup_strut**), move to insert approach position (**move_insert**), insert the strut (**insert**), and return to home (**move_final**). Input place **OK_insert** is added to transition **insert** so that the operator can authorize the insertion. Subsequent to each robot movement the path planner plans the appropriate path (**PP_insert** and **PP_final**). Lastly, the three subsystems are shutdown and the application returned to its starting state. Once the strut is removed from the assembly and returned to the strut rack the demonstration may be immediately repeated without restarting the system.

6 Concluding Remarks

The experiences of several researchers, particularly those in factory automation, have demonstrated the value of Petri nets for controlling distributed robotic systems. At the CIRSSE robotics laboratory we have had similar experience. The integration of PN controllers with robot and vision subsystem control software via a real-time, event-driven operating system has provided a very flexible environment for intelligent machine research.

Our use of Petri nets as an operator interface is more recent and less well developed. However, we are encouraged that the same tool, specifically Petri nets, can be used to analyze a complex, parallel, distributed system; to control that system; and to serve as an operator interface. The uniformity obtained by using one approach saves much effort otherwise spent converting between and reconciling different models.

Future work will concentrate upon the development of synthesis techniques and software tools for constructing large PN controllers from standardized subnet components. And, the role of the operator in supervising the intelligent machine will continue to be investigated.

Acknowledgement

We thank our many colleagues at CIRSSE who have contributed to the development of the IM computer architecture and PN controllers; the efforts of Profs. Alan Desrochers and Art Sanderson; and Greg Hamlin, Keith Nicewarner, Russ Noseworthy, Lance Page and Joe Peck are particularly appreciated. We would also like to thank NASA for funding this work under Grant NAGW-1333.

References

[1] R.Y. Al-Jaar and A.A. Desrochers, "Petri-nets in automation and manufacturing systems," *Advances in Automation and Robotics*, vol. 2, Greenwich, CT: JAI Press, 1989.

[2] P. Freedman, "Time, Petri nets, and robotics," *IEEE Trans. Robotics Automat.*, vol. 7, no. 4, pp. 417-435, August 1991.

[3] H. Lee-Kwang, J. Favrel, and P. Baptiste, "Generalized Petri net reduction method," *IEEE Trans. Sys., Man Cyber.*, vol. 17, no. 2, pp. 297-303, March/April 1987.

[4] D.R. Lefebvre and G.N. Saridis, "A computer architecture for intelligent machines," *IEEE Int. Conf. Robotics Automat.* (Nice, France), May 1992.

[5] T. Murata, "Petri nets: properties, analysis and applications," *Proc. IEEE* vol. 71, no. 4, pp. 541-580, April 1989.

[6] J.H. Park and T.B. Sheridan, "Supervisory teleoperation control using computer graphics," *IEEE Int. Conf. Robotics Automat.* (Sacramento, CA), pp. 493-498, April 1991.

[7] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, NJ: Prentice-Hall Inc., 1981.

[8] G.N. Saridis, "Toward the realization of intelligent controls," *Proc. IEEE*, vol. 67, no. 8, 1979.

[9] G.N. Saridis, "Intelligent machines: distributed vs. hierarchical intelligence," *Proc. IFAC/IMAC Int. Symp. on Distrib. Intelligence Systems* (Varna, Bulgaria), pp. 34-39, 1988.

[10] G.N. Saridis and K.P. Valavanis, K.P. "Analytical design of intelligent machines," *IFAC J. Automatica*, vol. 24, no. 2, pp. 123-133, 1988.

[11] F. Tendick, J. Voichick, G. Tharp, and L. Stark, "A supervisory telerobotic control system using model-based vision feedback," *IEEE Int. Conf. Robotics Automat.* (Sacramento, CA), pp. 2280-2285, April 1991.

[12] F. Wang and G.N. Saridis, "A coordination theory for intelligent machines," *IFAC J. Automatica*, vol. 26, pp. 833-844, 1990.

[13] F. Wang and G.N. Saridis, "A formal model for coordination of intelligent machines using Petri nets," *Proc 3rd IEEE Int. Intell. Control Symp.* (Arlington, VA), 1988.

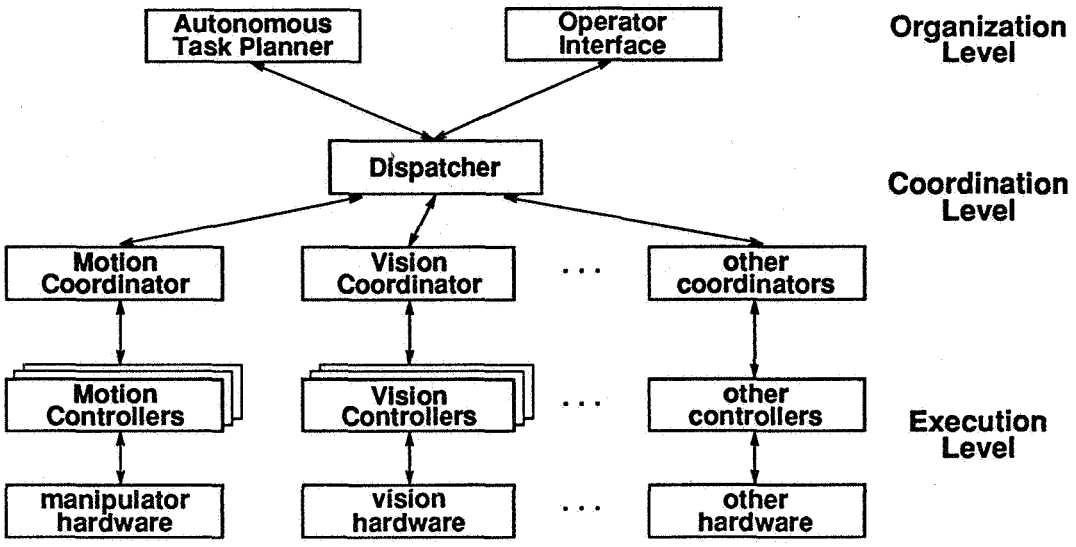


Figure 1: Hierarchical model of Intelligent Machine

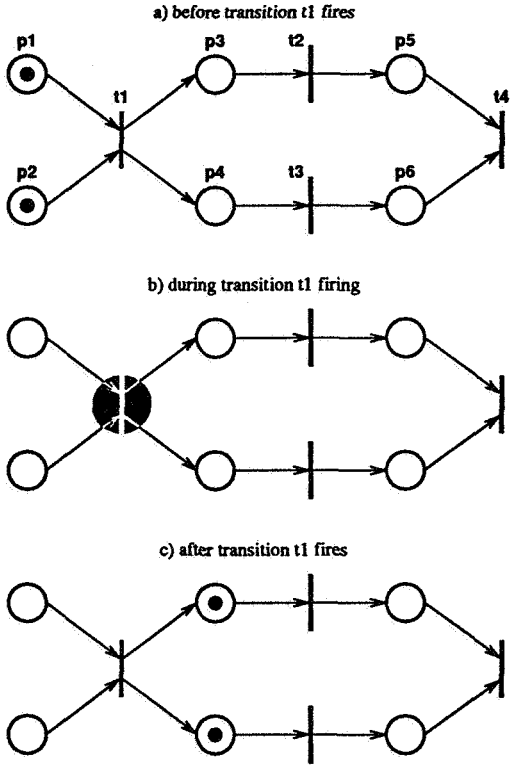


Figure 2: Example Petri net

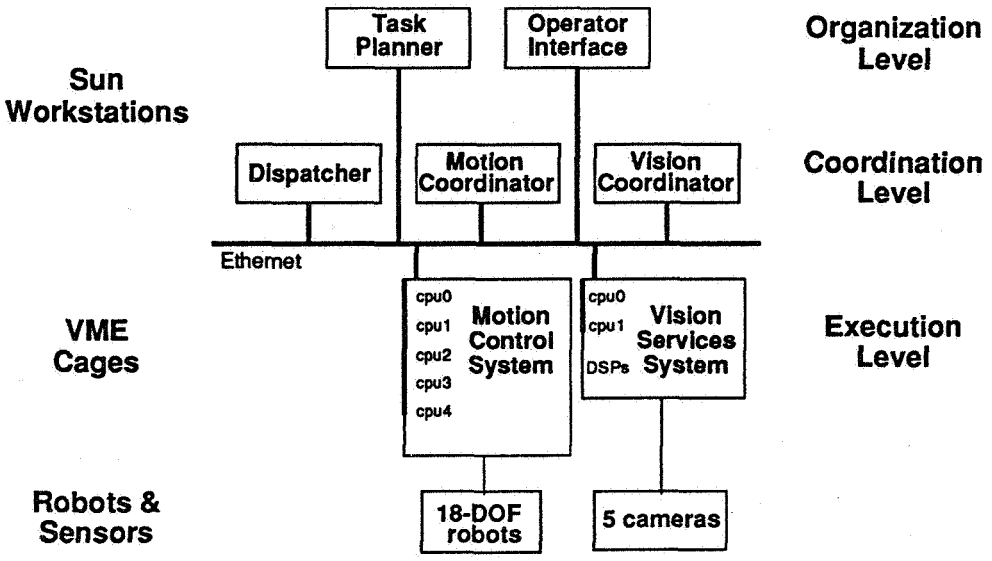


Figure 3: IM functions mapped to testbed hardware

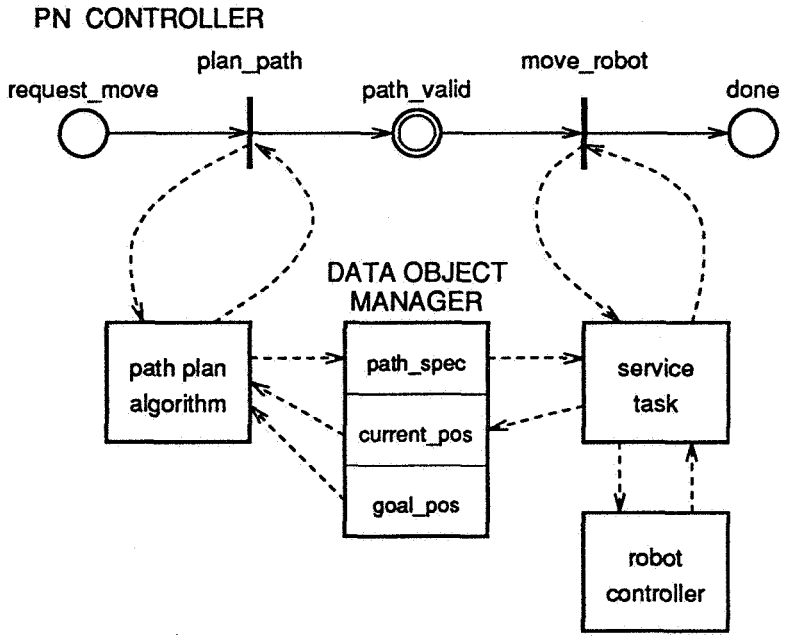


Figure 4: PN interface to CTOS tasks

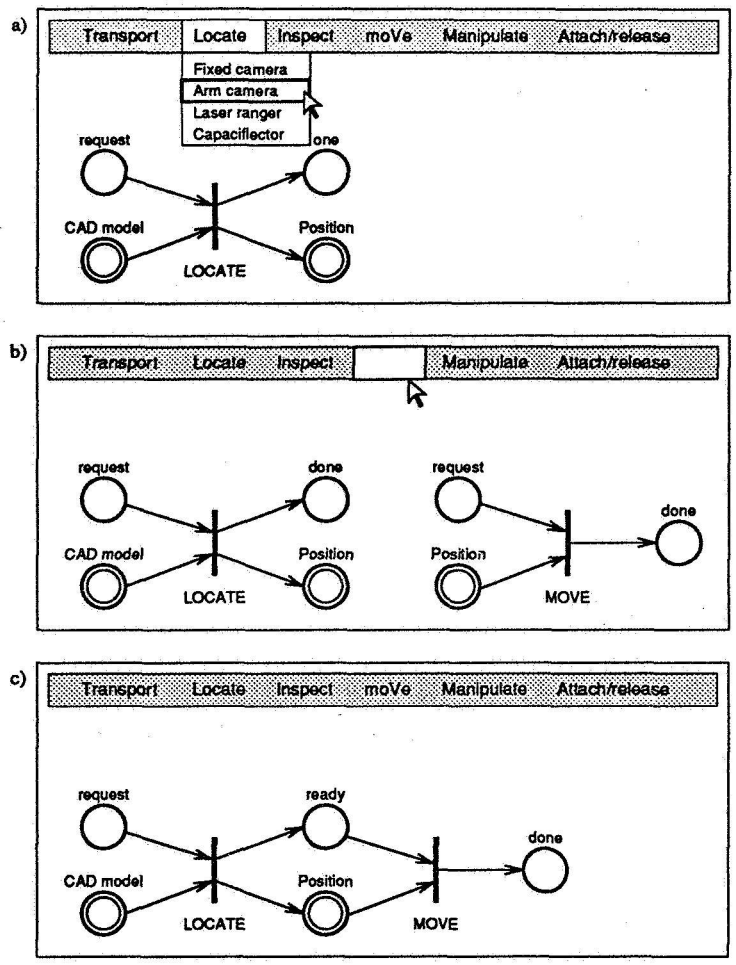


Figure 5: Concept of tool for building PN controllers

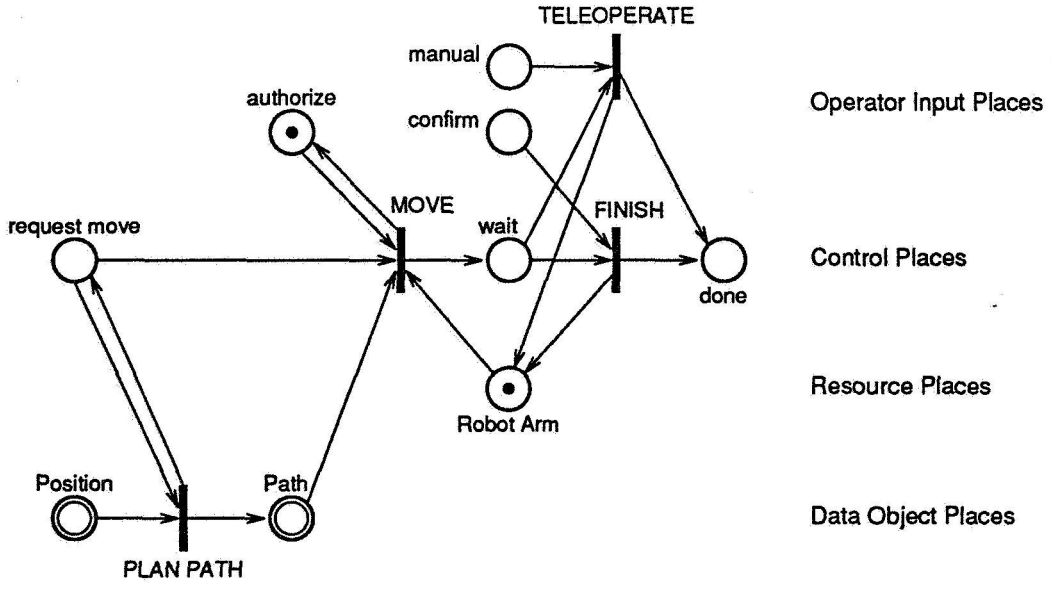


Figure 6: PN with operator interaction

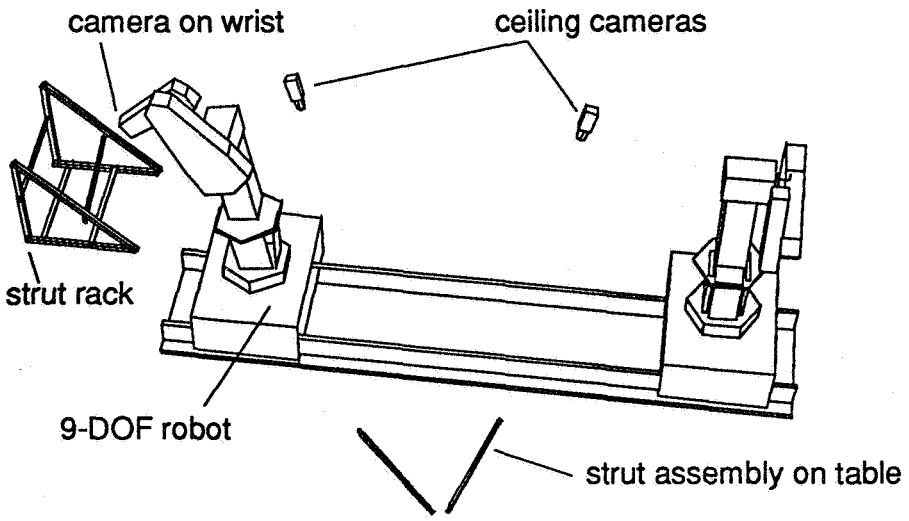


Figure 7: Diagram of Testbed

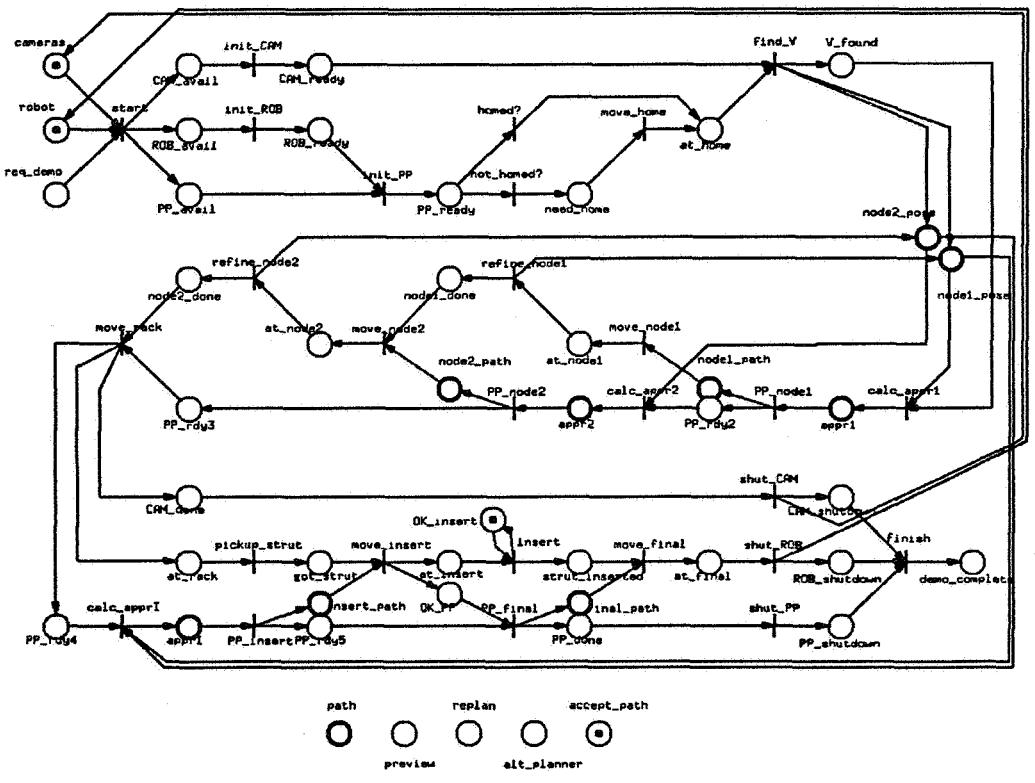


Figure 8: Dispatcher Petri Net Controller