UMIACS-TR-90-78 June, 1990
CS-TR-2483

Autoplan: A Self-Processing Network Model for an Extended Blocks World Planning Environment

C. Lynne D'Autrechy[1]
James A. Reggia[1,2]
Frank McFadden[1]

[2] Institute for Advanced Computer Studies and
[1]Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Self-processing network models (neural/connectionist models, marker-passing/message-passing networks, etc.) are currently undergoing intense investigation for a variety of information processing applications. These models are potentially very powerful in that they support a large amount of explicit parallel processing, and they cleanly integrate "high-level" and "low-level" information processing. However they are currently limited by a lack of understanding of how to apply them effectively in many application areas. This project is studying the formulation of self-processing network methods for dynamic, reactive planning. The long-term goal is to formulate robust, computationally-effective information processing methods for the distributed control of semiautonomous exploration systems, e.g., the Mars Rover. Our current research effort is focusing on hierarchical plan generation, execution and revision through local operations in an "extended blocks world" environment. This scenario involves many challenging features that would be encountered in a real planning and control environment: multiple simultaneous goals, parallel as well as sequential action execution, action sequencing determined not only by goals and their interactions but also by limited resources (e.g., three tasks, two acting agents), need to interpret unanticipated events and react appropriately through replanning, etc.

Acknowledgements: Supported in part by NASA award NAG1-885 and in part by NSF award IRI-8451430.

1.0 INTRODUCTION

The purpose of this research is to investigate the use of parallel processing models for plan generation, execution, monitoring, and replanning in the context of semiautonomous systems. In particular, the class of parallel processing models being studied here are *self-processing network models* incorporating message passing, marker passing, connectionist/neural modelling methods, etc. [Reggia and Sutton, 1988]. Information processing in these models occurs through numerous local and concurrent processes. Self-processing network models offer several potential advantages (highly parallel processing, adaptability, fault tolerance, clean integration of symbol-processing AI methods with numeric connectionist methods, etc.) if a way can be found to use them effectively.

The AutoPlan system (for "*Auto*nomous *Plan*ner") described within is a complete planning system (includes plan generation, execution, monitoring, and replanning) in which the behavior of the system is based solely on local operations. Plan generation is viewed as a *growth process* in which a goal-specific plan is generated from an underlying static*, goal-independent network. This growth process involves only local operations, mainly controlled through message passing but also by numeric spreading activation. As the dynamic goal-specific plan emerges from the underlying static network, local operations representing plan execution begin. These operations also involve both symbol-processing and numeric calculations; they commence even as plan generation continues and influence the plan generation process itself. As a result, plan generation and execution become almost inseparable processes.

The term *semiautonomous* as used above refers to a complex software-controlled system that is normally under human direction but which autonomously plans and carries out routine functions unless directed to do otherwise. Such a system should be able to function autonomously if necessary. Examples of semiautonomous systems include not only remotely controlled vehicles but other man-made systems such as nuclear power plants and factory automation systems. Our own work is particularly motivated by the Mars Rover, but it should be appreciated that the methods developed will be applicable to semiautonomous systems in general.

The division of responsibility between the plan generation and execution components of a planning system is determined by the predictability of the planning situation and hence the degree of reactivity needed by the system [Cheeseman, 1988]. In traditional planning systems the plan generation component generates an optimal and correct plan while the execution system simply implements the plan. These systems generally assume that the state of the world does not change during plan generation and execution; the predictability of the world is high and the reactivity needed in the system is low. In contrast, semiautonomous systems operate in a dynamically changing environment. Unexpected events can occur at any time and physical equipment failures are not uncommon. A planning system for a semiautonomous system must be able to react quickly to these types of environmental and internal changes. Since it is nearly impossible to plan ahead under these circumstances, the role of the plan generation system shrinks while the plan execution and monitoring systems take on more responsibility. The execution system must share in decisions such as what action to execute next and how to allocate limited resources. This is why the AutoPlan system uses an integrated plan generation and execution system, hereafter referred to as "the planner", in which the execution system has substantial responsibility for triggering and guiding planning and replanning.

*The underlying "static" network only changes when resources change or a fault or anomaly occurs, e.g., loss of a robot arm. In this case the relevant network components are added/deleted to/from the static network and subsequent plan generation and execution automatically adjusts to these changes.

Since plan failures are likely to occur in a dynamically changing environment, AutoPlan includes a plan monitor which takes appropriate actions in the context of unanticipated events. The plan monitor in the AutoPlan system uses strictly local operations to diagnose the cause of unexpected differences between the ideal and real world if possible, and then communicates this information to the planner so that it can react and replan accordingly.

This work investigates the use of competitive activation methods [Reggia, 1989] in resolving conflicts over limited resources. For example, only two robot arms may be available to do various tasks in a blocks world situation yet more than two tasks may require use of a robot arm. Each arm can only be allocated to one task at a time. Using competitive activation methods each task actively competes with the other tasks for the use of an arm. Previous work [Bourret, et al., 1989; Bourret, et al., 1990; Whitfield, et al., 1989; Goodall and Reggia, 1990] has shown competitive activation mechanisms to be an effective technique for resolving contention over limited resources.

This research work focuses on conjunctive planning and supports a distinction between goal achievement and goal maintenance. In conjunctive planning all goals must be accomplished although order is not always important. AutoPlan can be used in either situation, whether order is or is not important. The current work can also be expanded to encompass disjunctive planning. Goal maintenance means to "protect" a goal until its purpose is fulfilled (e.g., keeping a hand open until it has grasped a block). Goal achievement means that once a goal is achieved it is no longer important that the goal maintain a certain state (e.g., once a camera has taken a picture it has achieved its desired purpose).

The ultimate goal of this research is to implement and demonstrate a robust planning system for a planetary exploration vehicle (Mars Rover). Before this ambitious task is undertaken the ideas presented here are being tested in a simpler, dynamically changing blocks world environment we refer to as the "extended blocks world". This extended blocks world involves the two-dimensional surface of a table and the three-dimensional space above it. A limited number of robot arms are expected to concurrently (where possible) achieve multiple goals. Planning or plan execution may be interrupted at any time. The extended blocks world is motivated in part by the science experiments of the Mars Rover: individual goals (e.g., stack($\vec{loc}$, (*b1 b2*)) correspond to aspects of science experiments (e.g., "collect rock x"), a limited number of arms corresponds to constrained resources, etc. In addition to providing an initial critical evaluation of the concepts studied here, this work is investigating the ability of the Maryland MIRRORS/II connectionist simulator, hereafter referred to as MIRRORS/II, to model the described planning system and environment [D'Autrechy, et al., 1988]. Enhancements needed to MIRRORS/II to allow the development of AutoPlan and similar systems have been identified.

This report documents our progress in exploring the applicability of parallel processing models to plan generation, execution, monitoring, and replanning in the context of the dynamically changing extended blocks world environment. Section 2 discusses related and prerequisite work we and others have done. Section 3 provides a top-level system design. Sections 4, 5, and 6 describe each component of the system in more detail. Section 7 discusses the implementation of the system using the MIRRORS/II connectionist simulator and necessary enhancements to MIRRORS/II. Finally, Section 8 is a discussion and summary of the work described within. Extensive design details of the plan generation and execution portion of the system can be found in Appendix A while details of the plan monitor and diagnosis component can be found in Appendix B.

2.0 BACKGROUND AND RELATED WORK

Some related work has been done in the area of using parallel processing models to do planning, resource allocation, and scheduling. However, the work that has been done in applying self-processing network techniques to planning seems to be the most limited. While semiautonomous systems operate in a dynamically changing environment, most of the models described below assume that the state of the world remains static during plan generation and execution and that plan generation and execution are totally separate processes. Also, the AutoPlan system described in this report combines components of planning — plan generation, execution, monitoring, and replanning, into a cohesive integrated system. Unlike more traditional planning systems which use assertions to model the state of the world, AutoPlan represents each physical entity of the extended blocks world (i.e., arms, grid locations, and blocks) and its state as part of the self-processing network used for planning. Hence, the state of the ideal world contributes to the planning process.

The AutoPlan system is closest to and most influenced by Sliwa's "behavioral networks." Building on her earlier work in telerobotics [Sliwa and Soloway, 1987], Sliwa has proposed behavioral networks as a methodology for integrating features of low-level robotic behavior such as activation and sensing with other more intelligent activities such as planning, scheduling, and learning [Sliwa, 1989]. Such a system must be able to operate in a dynamically changing environment and handle resource allocation conflicts. Behavioral network models are a hybrid of classical control techniques, artificial intelligence planning methods, and connectionist approaches. A behavioral network itself can be thought of as an acyclic directed graph whose nodes represent specific functions, or behaviors, of an intelligent system, with two-way links which propagate information including functional parameters and weights. Behaviors at higher levels in this hierarchy decompose into descendent behaviors at lower levels in the hierarchy. A prototype behavioral network is currently being developed and tested. AutoPlan is similar in spirit but differs in detail from behavioral networks. AutoPlan also models an ideal environment (blocks, arms, etc.) as active network components that contribute to the plan generation process.

A model of generating navigational plans for a ship in a two-dimensional space has been proposed by Miyata [Miyata, 1988]. His planner assumes that the environment remains static during plan generation. A network with seven layers of nodes is used: one layer represents the desired state of the environment (the input layer), another represents the action plan, yet another represents the predicted state of the environment, two more layers are layers of hidden units, one is between the desired environment and action plan layers and one of which is between the action plan and predicted environment layers, and the remaining two layers are context layers, each of which is connected to both hidden layers. A two-phase supervised learning algorithm is used to train the network. First the network uses an error backpropagation technique to learn to associate certain actions with certain predicted environmental states, and then the network uses the same technique to learn to associate certain desired environmental states with certain actions. Once learning is complete a goal, represented as a pattern of activity on the input environment units, is presented to the system. The system then goes through an iterative plan generation and refinement sequence until the plan generated is able to achieve the goal. The plan is represented by rows of two units, one row for each time step. The activation value of units on the right represent the amount the ship should accelerate to the right during that time step. Similarly, the units on the left represent the amount of acceleration to the left for a specific time step.

Whitehead and Ballard have proposed a method for executing stored plans [Whitehead and Ballard, 1988]. A stored plan is one which may be applied to many situations and therefore can be instantiated with different situation-specific parameters. A plan is viewed as a hierarchy of actions which can be represented by an acyclic graph. Nodes higher up in the

hierarchy represent abstract actions composed of the more detailed actions at lower levels of the hierarchy. A sequential network with an architecture like that proposed by Jordan [Jordan, 1986] is used to do plan execution. The network learns to associate one step in the execution of the plan with the next step in the execution sequence.

A model for doing plan execution has recently been developed which uses a localized parallel processing technique [D'Autrechy and Reggia, 1989a]. The system starts with an existing "Spaceworld" plan, a plan for a simplified model of the Voyager spacecraft which photographed Jupiter, Saturn, and their satellites [Vere, 1988]. It then uses a marker-passing paradigm to execute the plan. The system simulates a parallel execution environment; this allows independent nodes in the plan to be executed concurrently while other nodes which are dependent on each others' execution completion can be executed sequentially. Both non-hierarchical and hierarchical plans have been executed successfully. Based on the success of this earlier system, the AutoPlan system uses marker-passing to distribute symbolic information during the integrated plan generation and plan execution process.

Hendler has developed a hybrid planning system called SCRAPS which uses a combination of traditional AI methods and marker passing [Hendler, 1988]. He augments a planner by adding a parallel marker-passing component which identifies relevant paths in an associative network. These paths are then inspected for certain types of information which, if present, cause the planner to modify its current plan. SCRAPS differs from AutoPlan in that is uses a self-processing network as an adjunct to traditional AI planning methods rather than as the central planning mechanism. Hendler, a leading advocate of hybrid AI and connectionist systems, has subsequently investigated other approaches that are complementary to the approach taken in AutoPlan [Hendler, 1989].

Maes has implemented a model of action selection (plan execution) for autonomous systems [Maes, 1989]. Her system starts with a pre-wired non-hierarchical network of primitive competence modules (nodes). It does not do any plan generation. Modules have excitatory connections to other modules which help satisfy their preconditions and modules which are successors to themselves. Modules also have inhibitory connections to other modules with which they are in conflict. Strictly numeric spreading activation is used to do action selection. Modules receive input from the goals to be achieved, the current state of the environment, and other modules. The system has a number of global parameters which can be used to adjust the system's responsiveness to goal-relevance versus situation-relevance, adaptability versus speed, and other such trade-offs. While the computations which take place are primarily local, there are some global computations which preclude this model from being strictly a connectionist model in which only local operations are performed. The system is also unable to handle variable binding.

The system proposed by Maes is able to adapt to changes in the goals being achieved and/or to changes in the environment. The numeric input to the system based on the goals and environment informs the system of the changes. Maes claims that the system can respond successfully when a competence module fails (e.g., a hand drops a board) but, without a plan monitor, it is difficult to see how such failures are detected or used by the system.

Connectionist models have been studied to determine how to make the best use of limited resources while still satisfying as many constraints as possible [D'Autrechy and Reggia, 1989b]. The models discussed below, one for satellite transmission scheduling and one for camera tracking, demonstrate the viability of using competitive activation methods to allocate limited resources successfully.

In the satellite-antennas communication scheduling scenario, several low-level satellites gather information as they orbit the earth. During each revolution a satellite can broadcast its

accumulated information to an antenna only during the short time period within which it is visible to that antenna. Since a satellite may gather information during a single revolution that exceeds the time within which that satellite is visible to a single antenna, its broadcasting might need to be split into messages to several antennas. It is assumed that two satellites cannot transmit to the same antenna at the same time, and that satellites have varying priorities according to the importance of their information. A successful prototype connectionist model for solving satellite-antennas communication scheduling problems has been devised [Bourret, et al. 1989]. The goal of this model is to generate communication schedules that maximize priority-weighted transmission time. Nodes representing each satellite effectively compete for available time slices on appropriate antennas. Preliminary testing suggests that this approach may be used effectively [Bourret, et al., 1989, 1990].

Another resource allocation problem considered is that of tracking and photographing designated targets with cameras on a spacecraft [Whitfield et al., 1989; Goodall and Reggia, 1990]. The scenario, simplified from real life, involves three mobile cameras that are available to photograph numerous designated locations as the spacecraft passes over them. The problem to be solved is for the cameras to move so as to track incoming target locations; the cameras are to automatically position themselves to take a photograph of as many targets as possible just before the targets pass under the spacecraft. The cameras "compete" for targets to photograph, and a competitive activation mechanism is used to implement this competition. Two sets of experimental tests have been done demonstrating the viability of using competitive activation mechanisms in this fashion to allocate resources effectively in dynamic environments [Whitfield et al., 1989; Goodall and Reggia, 1990].

3.0 TOP-LEVEL SYSTEM DESIGN

The AutoPlan system has a number of features:

- self-processing network model - information processing is strictly based on local operations
- complete planning system includes plan generation, execution, monitoring, and replanning
- reactive planning - the planning system reacts to changes in a dynamically changing environment and replans accordingly
- resource-limited planning - competitive activation methods are used to allocate limited resources
- conditional planning - some primitive actions are only executed under certain environmental conditions
- deferred planning - decisions regarding the execution of conditional parts of a plan, allocation of limited resources, and the order of execution of the actions of the plan are deferred until the time during plan execution when the decisions must be made.
- hierarchical planning - a plan in the form of a hierarchy is generated; nodes at the top-level of the hierarchy are the most abstract while the lowest level is composed of primitive action nodes which, when executed, change the state of the world.

Many of these features are innovations in the context of self-processing network models.

Extended Blocks World

We refer to the dynamically changing environment in which AutoPlan operates as an "extended blocks world." It differs from traditional blocks worlds in the following respects: the two-dimensional table has a finite size, the space above the table is treated as a three-dimensional grid and hence has specific spatial locations (e.g., (2 2 1)), multiple robot arms may be present thereby allowing multiple goals to be achieved in parallel, and anomalous conditions may occur. See Figure 1. The extended blocks world is represented explicitly as part of the self-processing network. Each entity in the blocks world (e.g., grid location, arm) is represented by its own node in the network. This differs from other planners which represent the world as a set of assertions (e.g., (on a b)).

The Plan

Goals are states of the world which a plan tries to achieve. A *plan* represents one possible sequence of steps that can be taken to achieve a given set of goals. The plan generated by AutoPlan is a hierarchy of *actions* where each lower level in the plan hierarchy represents a more detailed level of abstraction. The lowest level actions are the primitive actions which when executed actually cause changes in the state of the world. Actions have *preconditions*, conditions which must be true in order for the actions to execute and *effects*, changes in the state of the world as a result of the execution of the actions.

In the plans generated by the AutoPlan system, some of the connections from an action to its subactions or child actions are conditional. See Figure 2. These conditional links allow AutoPlan to do conditional planning based on the current state of the world when it is time to execute the subactions at the end of the conditional links. For example, suppose the pickup(*a, b*) action connects conditionally (if another block is on top of block b) to a moveblock(*a, b', $\vec{loc}$*) action where *b'* is a block on top of block *b*. The moveblock action only gets executed if another block is on top of block *b* at the time the planner is ready to execute the child actions of the pickup action.

The Planning System

Conceptually one can think of the AutoPlan system as consisting of four components: plan generation, execution, monitoring, and replanning. In reality there are only two tightly-coupled components: the planner, an integrated plan generation and execution module which also handles replanning, and the plan monitor which notices differences between the ideal state of the world, as maintained by the planner, and the real world, diagnoses the cause(s) of the observed differences, and communicates this information to the planner so that it can react and replan accordingly. See Figure 3.

Thus far this discussion of the planning system has not differed from one which might be found in a paper describing a more traditional planning system with centralized control. However, in a self-processing network model one can no longer think of each of the planner and monitor as separate but whole software modules. Instead one must think of the planner and monitor as overall behaviors which emerge from the local computations taking place between the nodes in a self-processing network. Each node in the network executes some set of functions which implement the planner and monitor functionality *for that node*. A separate piece of software which implements the planner or monitor for *all* the nodes in the system does not exist.

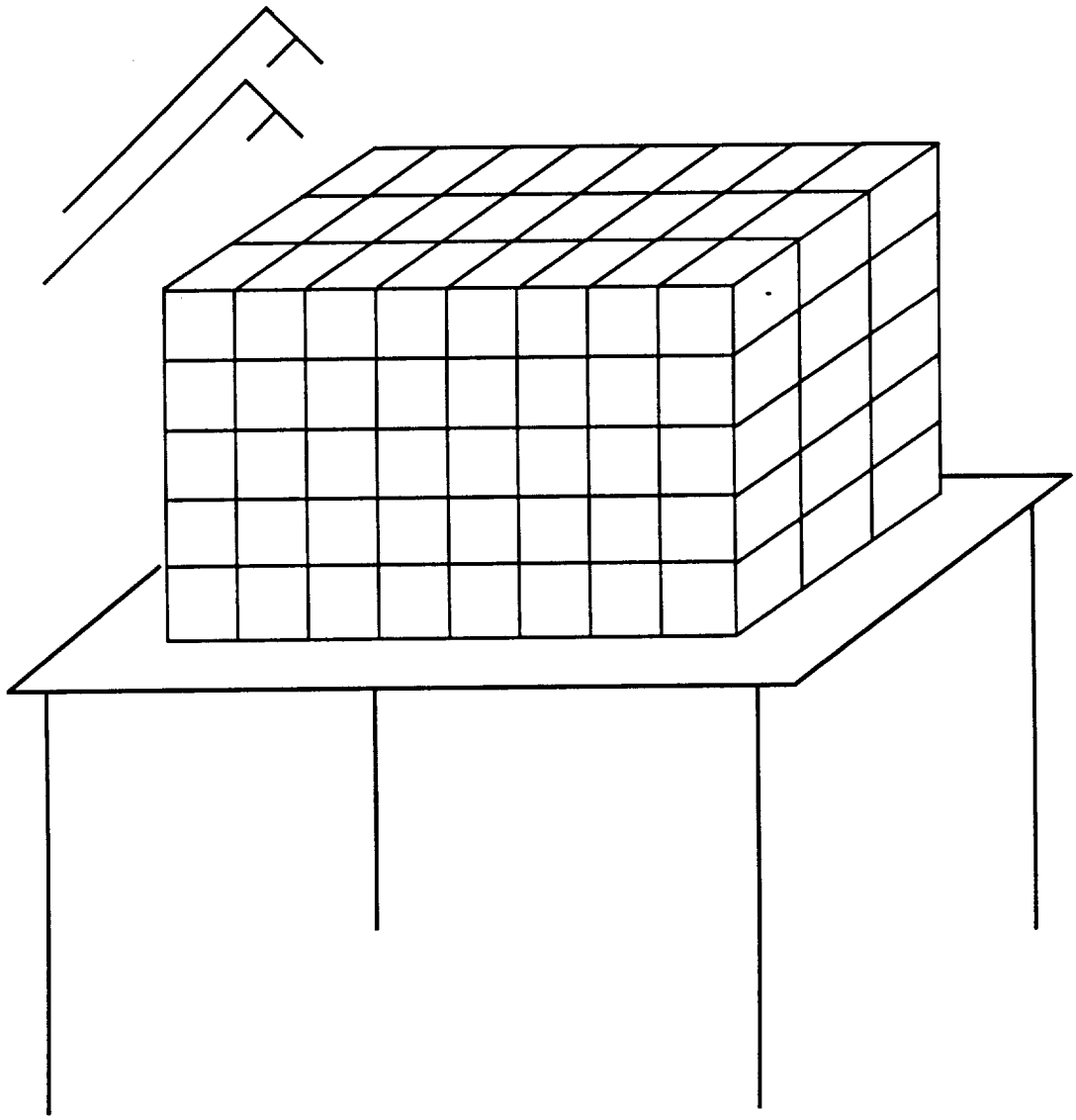
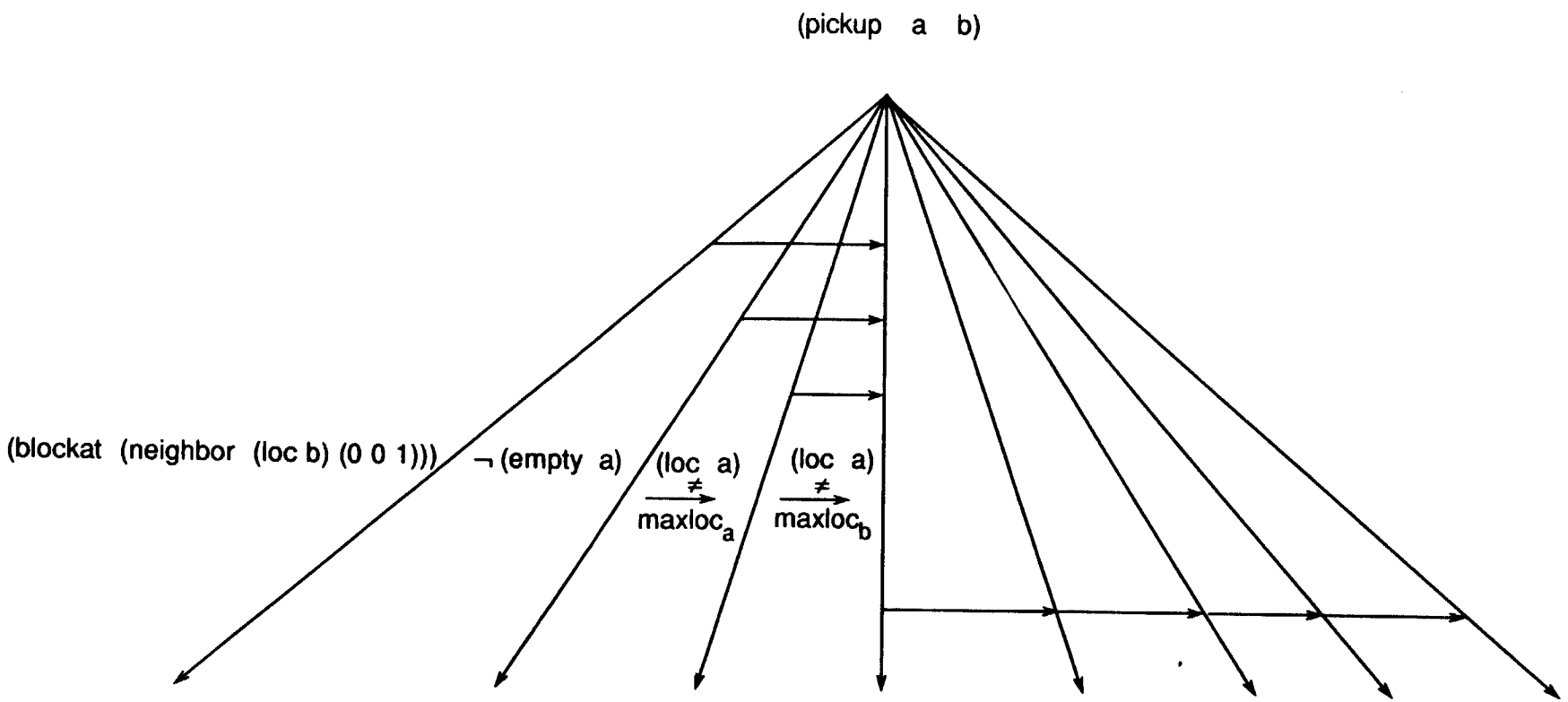


Figure 1: 3-D Extended Blocks World Space



(moveblock a′ c′ $\overrightarrow{loc_u}$)(putdown a c $\overrightarrow{loc_u}$)(up a $\overrightarrow{maxloc_a}$)(movearm a $\overrightarrow{maxloc_b}$)(open a)(down a $\overrightarrow{loc_b}$)(close a)(up a $\overrightarrow{maxloc_a}$)

c′ = (blockat (neighbor (loc b)(0 0 1)))

c = (holding a)

Figure 2: Conditional Planning

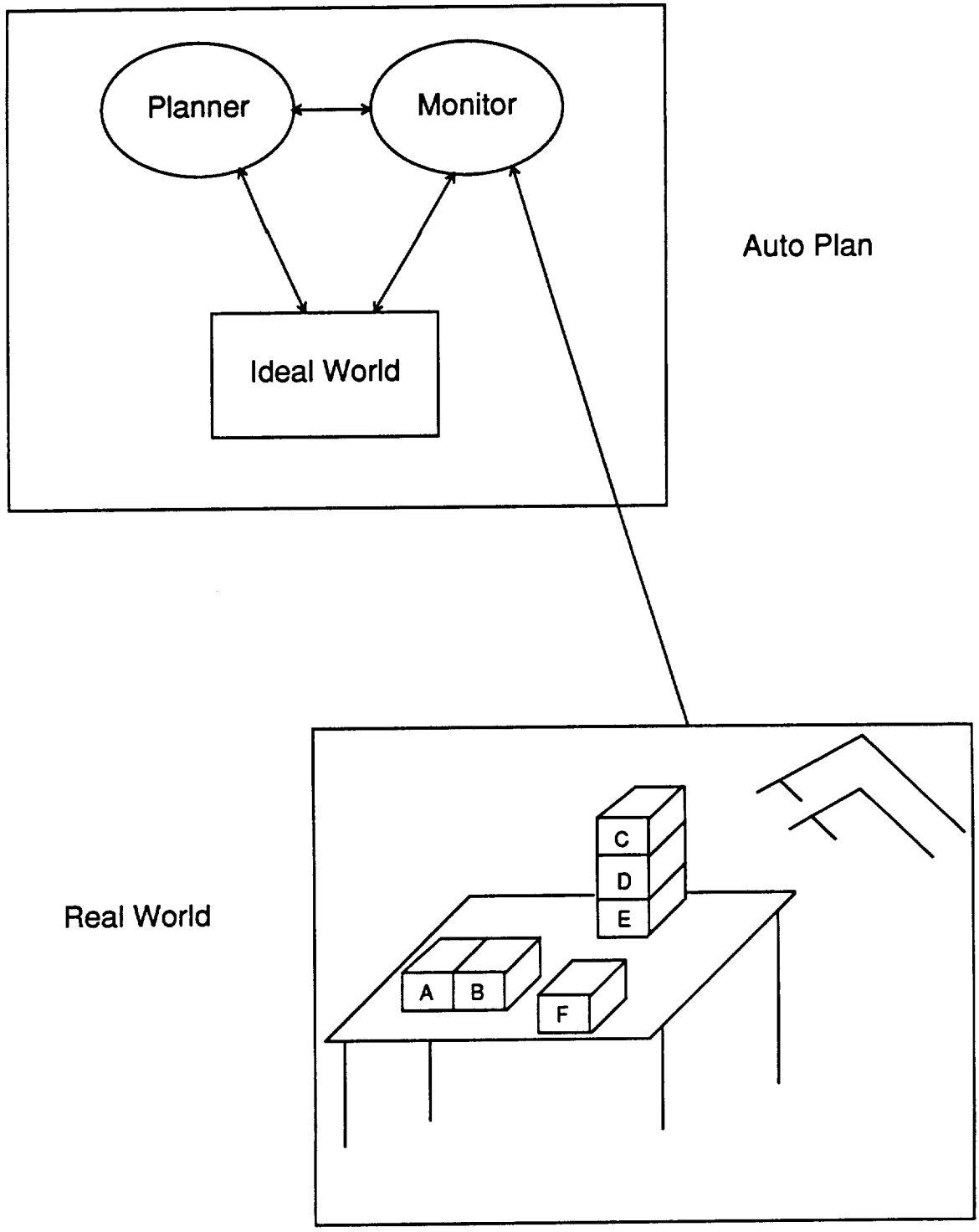


Figure 3: The AutoPlan System

4.0 PLAN GENERATION AND EXECUTION

In traditional planning systems planning is split into two separate and sequential processes — plan generation and plan execution. First the planner generates a plan that is typically an "optimal" sequence of steps used to achieve a given goal. Then the planner executes this plan. The problem with this division of responsibilities is that during the plan generation phase the state of the world may change thus rendering the plan useless.

In order to plan in a dynamically changing environment, the processes of plan generation and execution must be integrated. The desirability of an optimal plan decreases and it becomes much more desirable to have a system which can adapt to changes in the environment during the planning process. AutoPlan has an integrated plan generation and plan execution process also referred to as the planner. In fact, the plan generation and plan execution process is actually a behavior which emerges as the result of functions executed in parallel by nodes in a parallel processing network.

An Example

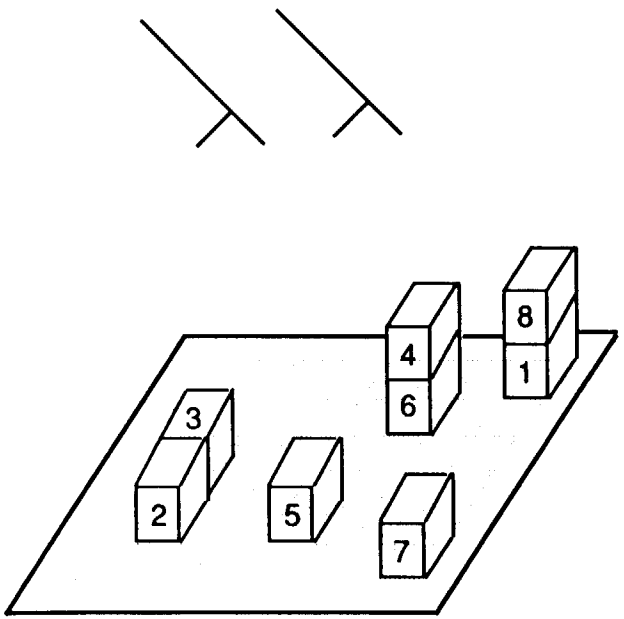
At this point it would be useful to look at an example. Suppose the top-level goals to be achieved by the planning system are (stack (1 1 1) (b2 b3 b4)) and (align (3 3 1)(1 0 0)(b5 b6)). Assume that the blocks world system has two arms a1 and a2. All the dynamic nodes for the blocks, grid locations, and arms sets already exist. Only the dynamic action and binding nodes remain to be created at run time by the planner. See Figures 4 and 5.

Plan generation is a growth process controlled by message passing. First, each static action node which corresponds to a top-level goal (e.g., stack and align) is sent a message telling it to create a dynamic instance of itself. The message includes the values for the variables of the action nodes. Figure 6 illustrates this just for the single goal (stack ...); other goals, such as align, are generating separate growth. The dynamic action node (stack) is passed pointers to the static action node (stack), the descendents of the static action node (moveblock), and the static block binding node. Once the dynamic node has been created it uses these pointers to connect back to its static counterpart, the descendents of the static action node, and the static block binding node. See Figure 7.

Each newly created dynamic action node still needs to do variable binding so it sends a message to the static block binding node telling it the blocks to which it needs to connect. The block binding node then creates an instance of itself for each block to which an action node needs to connect. See Figure 8. Each new dynamic block binding nodes points back to its static counterpart and to the static block node to which the static block binding node is also connected. The static block binding node sends a message back to the dynamic action node containing pointers to the new dynamic block binding nodes. See Figure 8. Upon receipt of this message, the dynamic action node deletes its connection to the static block binding node and makes new connections to the dynamic block binding nodes. Each block binding node sends a message to the static block node requesting the pointer to a specific dynamic block node. The static block node then sends back the pointer to the requested dynamic block node. See Figure 9. Only the pointer for block $b3_1$ is illustrated. The dynamic block binding node then deletes its connection to the static block node and, using the pointer it just obtained, makes a new connection to the dynamic block node. See Figure 10. A similar process occurs for the arm binding and arm nodes, and the grid location binding and grid location nodes but is not illustrated here.

Simultaneously, the align action node and the corresponding binding nodes for the align action node are created. No binding node is created for the second argument $\vec{d}$ = (1 0 0), the

Initial State



Goal State

(and (stack (1 1 1)(b2 b3 b4))
(align (3 3 1)(1 0 0)(b5 b6)))

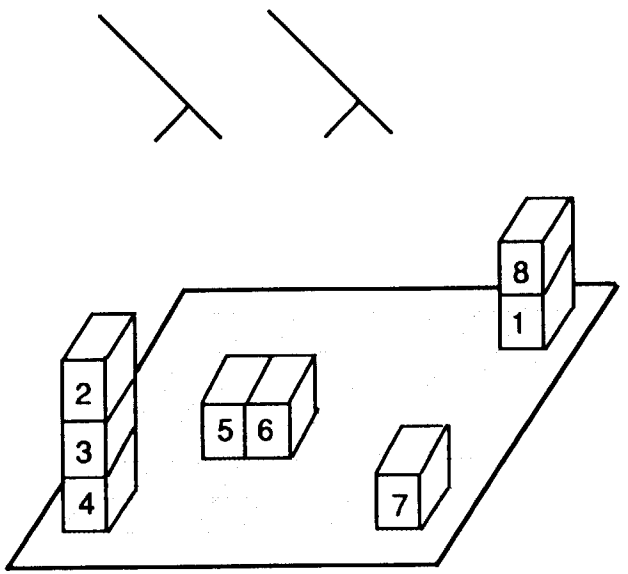


Figure 4: Example Initial and Goal States

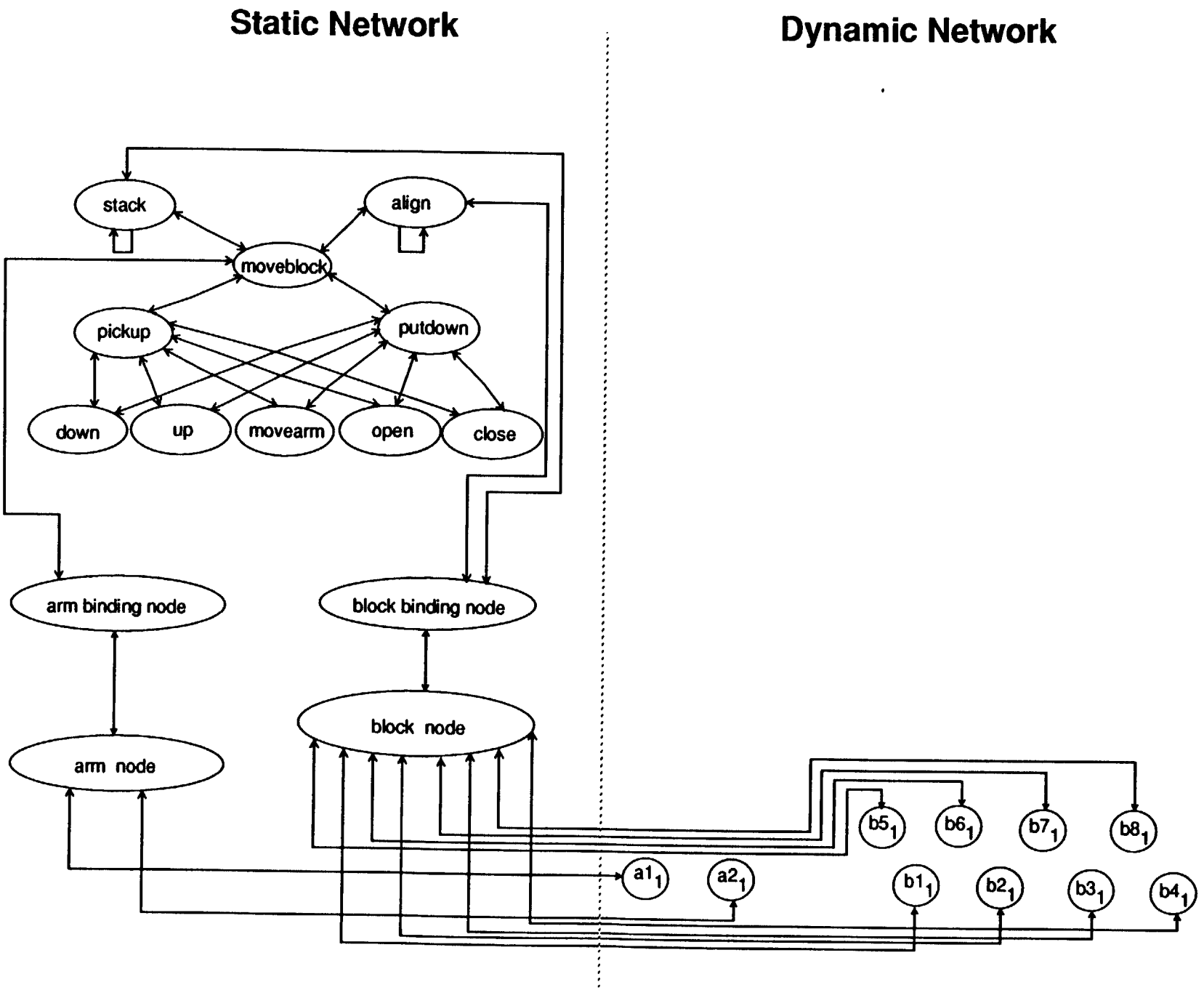


Figure 5: The Static and Dynamic Networks before Runtime

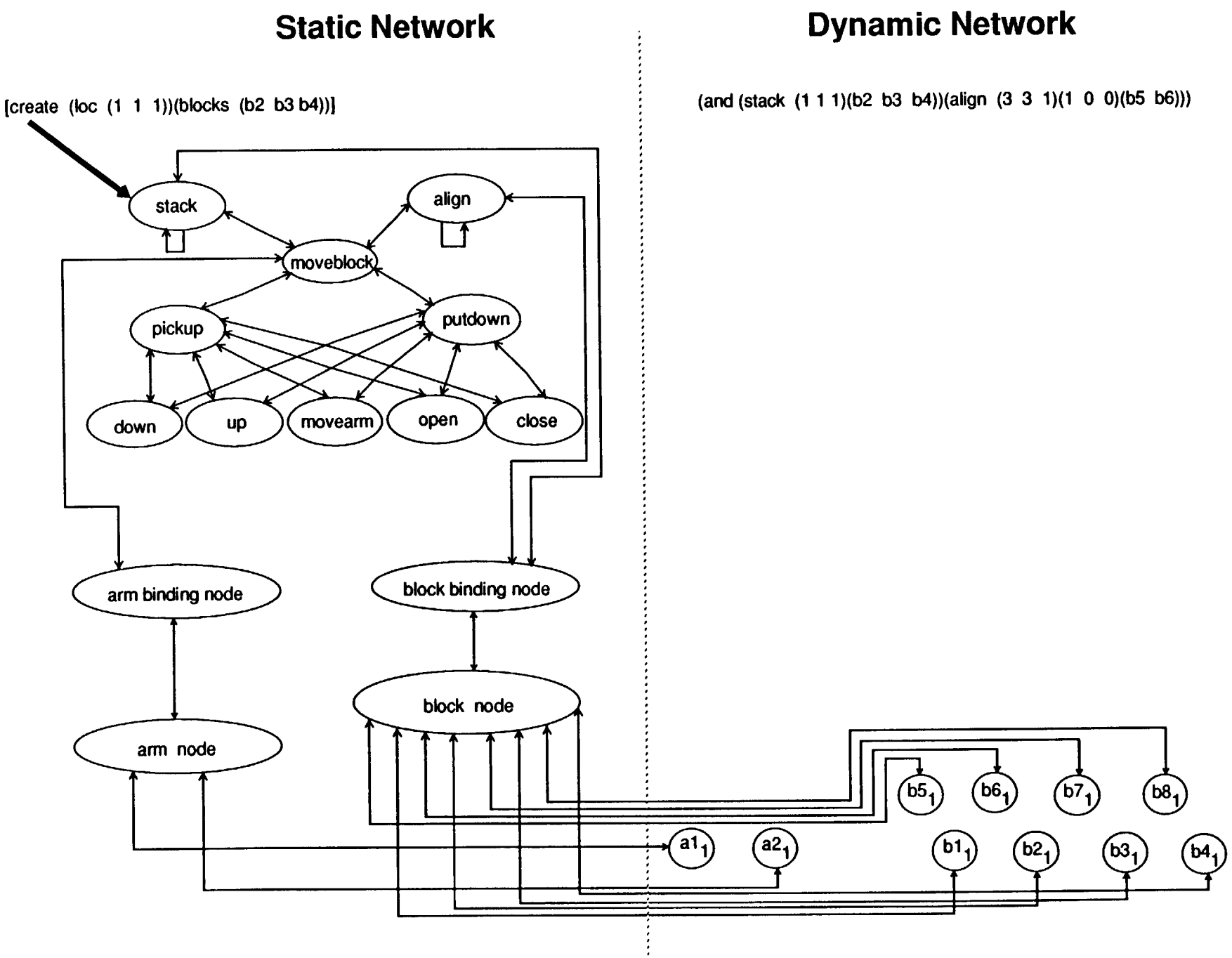


Figure 6: Creation of the Top-Level Actions

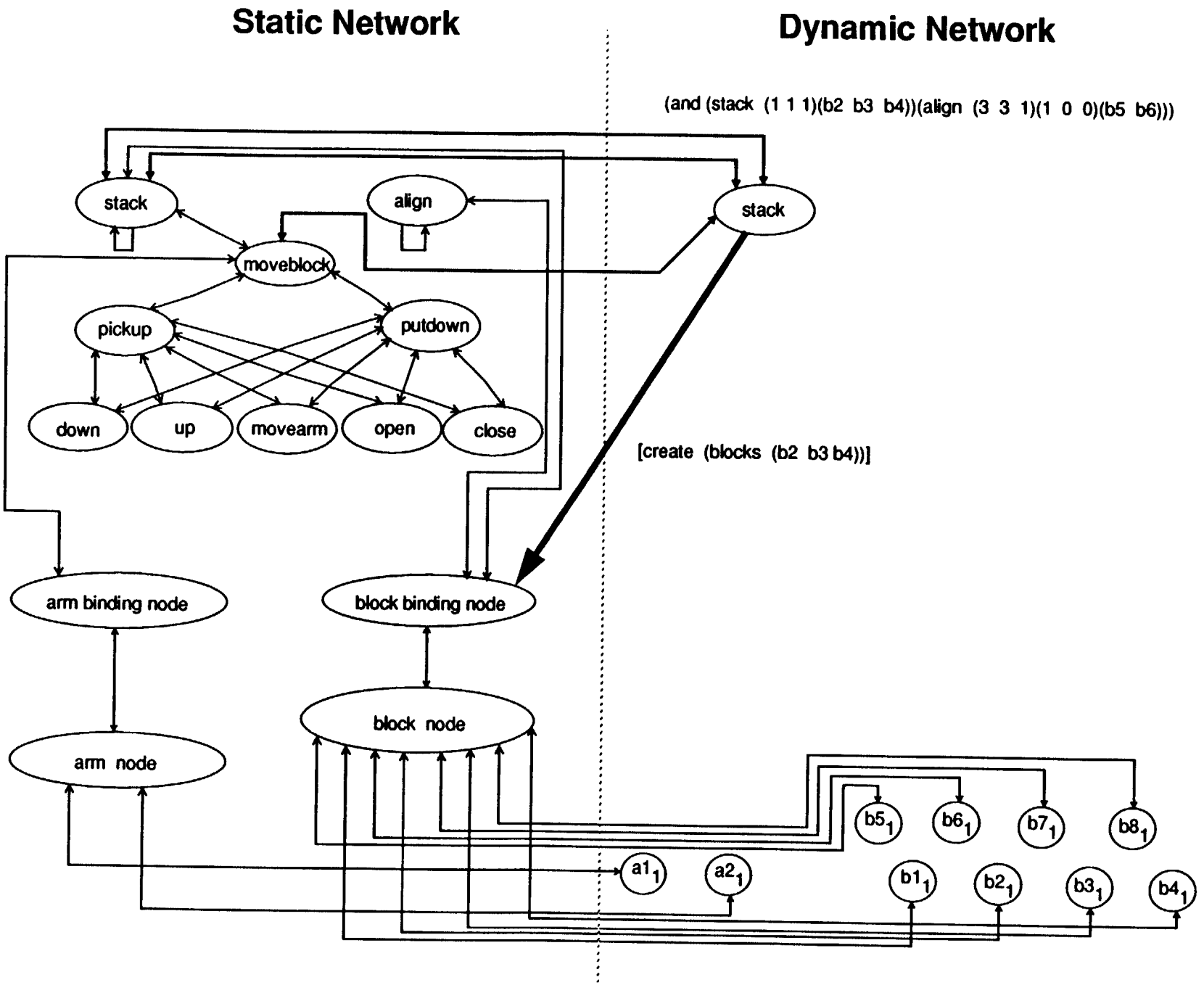


Figure 7: Creation of the Block Binding Nodes

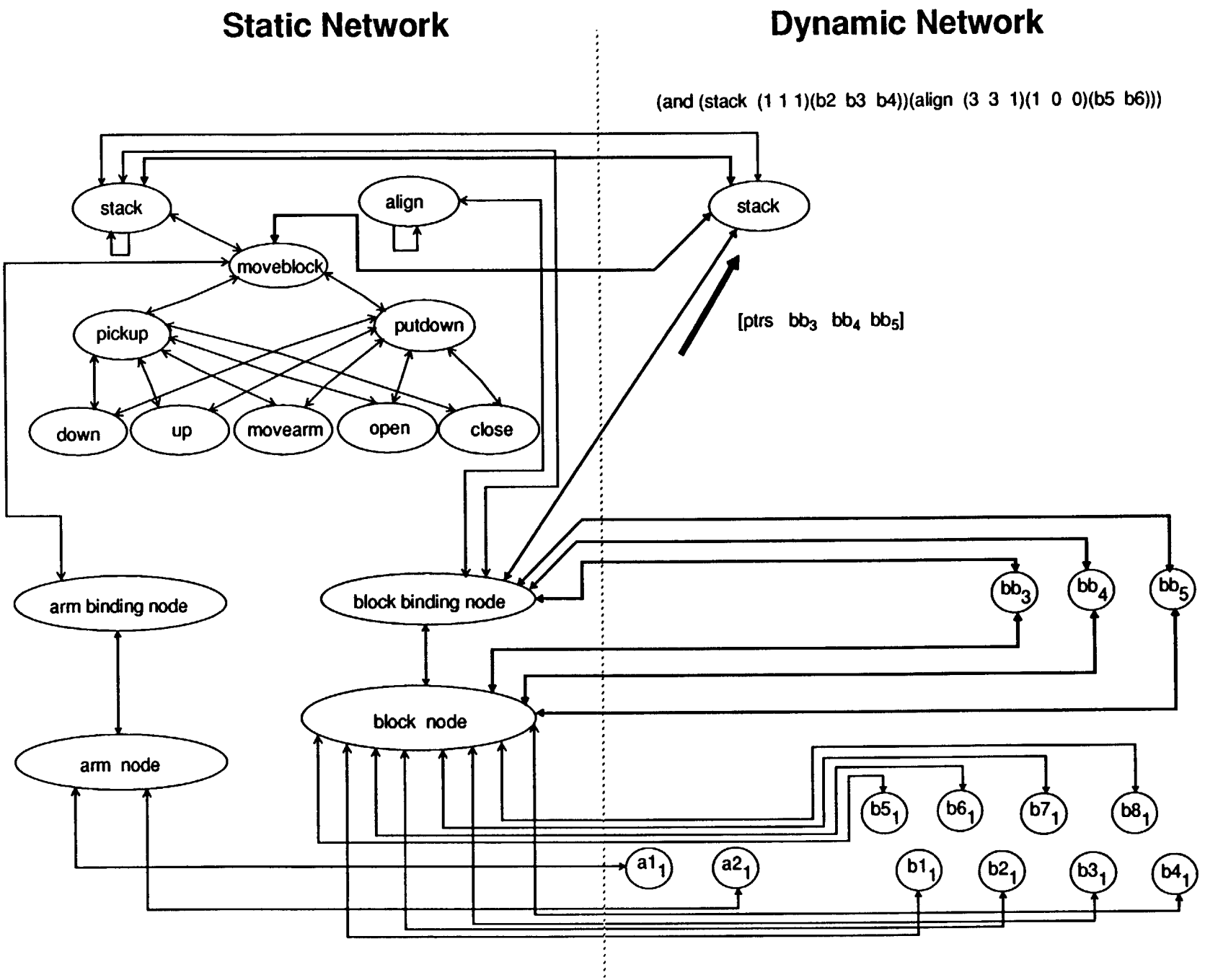


Figure 8: Creation of the Block Binding Nodes

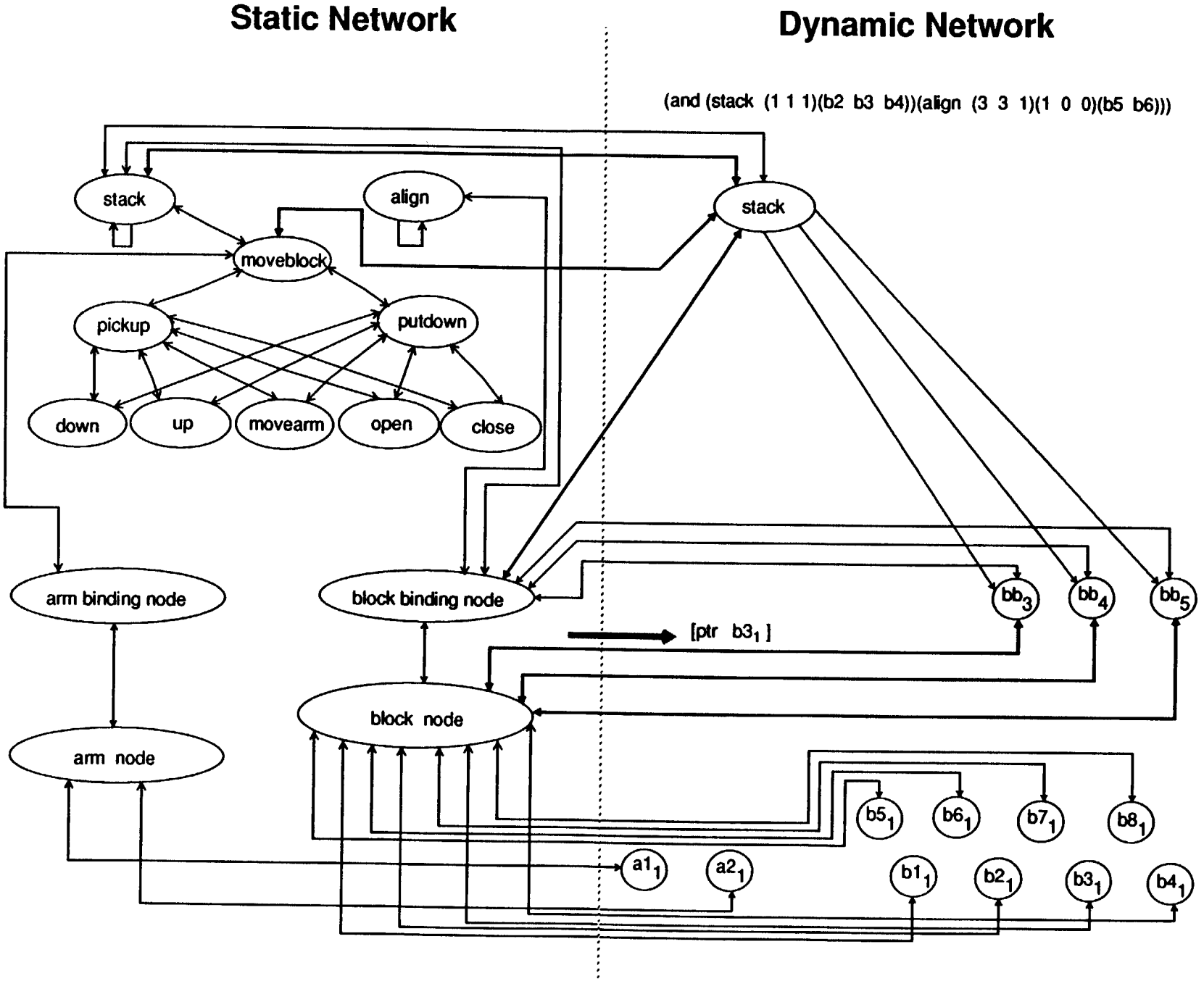


Figure 9: Creation of the Block Binding Node Connections

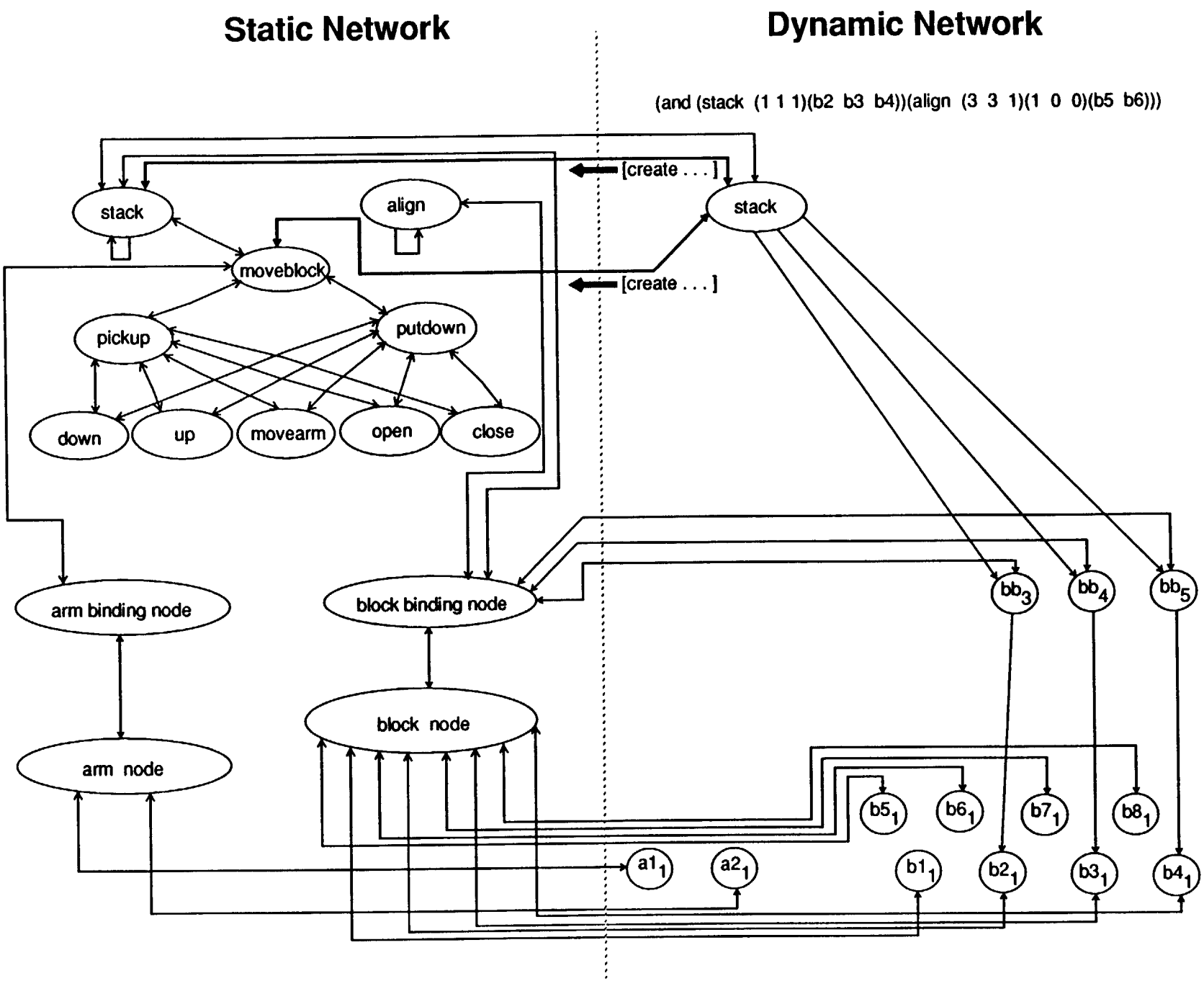


Figure 10: Creation of the First-Level of Child Actions

direction of the alignment since the direction is not a resource for which the align node needs to compete. The direction variable slot of the align action node is simply set to (1 0 0).

Next the stack and align action nodes each receive an external input of 1.0. They in turn send output to each binding node to which they are connected. The binding nodes receive this input and update their own activation values. During the next iteration the action nodes send output to the binding nodes and the binding nodes send output to the action nodes and the various resource nodes to which they are connected. Then all the nodes update their activation values. The following iteration is essentially the same except that the resource nodes now have activation which they can send to the binding nodes. This process continues until a binding node becomes fully activated. At this point the binding node takes the resource to which it connects out of the competition.

One might think that at this point the binding node has done its job and can be pruned from the network. This is not the case. A binding node remains part of the network until the action node to which it is connected finishes executing or releases it for some other reason. The binding node is still needed for two reasons. First, it may be needed later if replanning is necessary before the action node (and its descendents) have finished executing. Second, the action node frees a resource once it has finished with it by communicating to the resource via the binding node to which they are mutually connected.

The child action nodes of a parent action node are not generated until all the variables of the parent action node are bound. It is inefficient to generate child actions until one is sure the parent action is able to obtain the resources it needs since most child actions inherit these resources from their parent. Once the stack action has its variables, $\vec{loc}$ and *blocks*, bound it sends a message to the static moveblock action node and stack action node, children of the existing static stack action node telling them to create dynamic instances of themselves. The parent stack node is then connected to these nodes and a lateral sequence connection from the child moveblock node to the child stack node is created. The moveblock node inherits bindings for its *b* and $\vec{loc}$ variables from the parent stack node. However, it still needs to bind an arm node. The child stack node inherits all its bindings from the parent stack node so no new binding nodes need to be created for it. Simultaneous to the generation of child nodes for the top-level stack action, analogous variable binding and action nodes are generated for the top-level align action node. See Figure 11.

Once the parent action nodes are no longer competing for resources, all their activation is sent to their child action nodes. The moveblock child node uses that incoming activity to compete for an arm node. It competes against a moveblock node which is a child of the align node.

In parallel to the arm node competition, the children of the child stack action node are generated since the child stack action node has all its variables bound. Once the children are created the child moveblock node of the child stack node also enters into the competition for the arm nodes. However, it does not compete as effectively as the other two moveblock nodes because its parent stack action node is not as active as the stack node's sibling moveblock node. This is because the stack node is not receiving much lateral input from the moveblock node and therefore cannot get as active as the moveblock node. See Figure 12.

As the process continues, child action nodes for pickup and putdown nodes are created. Some of the child action nodes of the pickup or putdown nodes are only created under certain conditions. The state of the world is checked when it is time to generate each child and then some of the children are only generated if the conditions necessary for their creation are currently present. By delaying the generation of children for as long as possible, the plan generated is more appropriate to the current state of the extended blocks world.

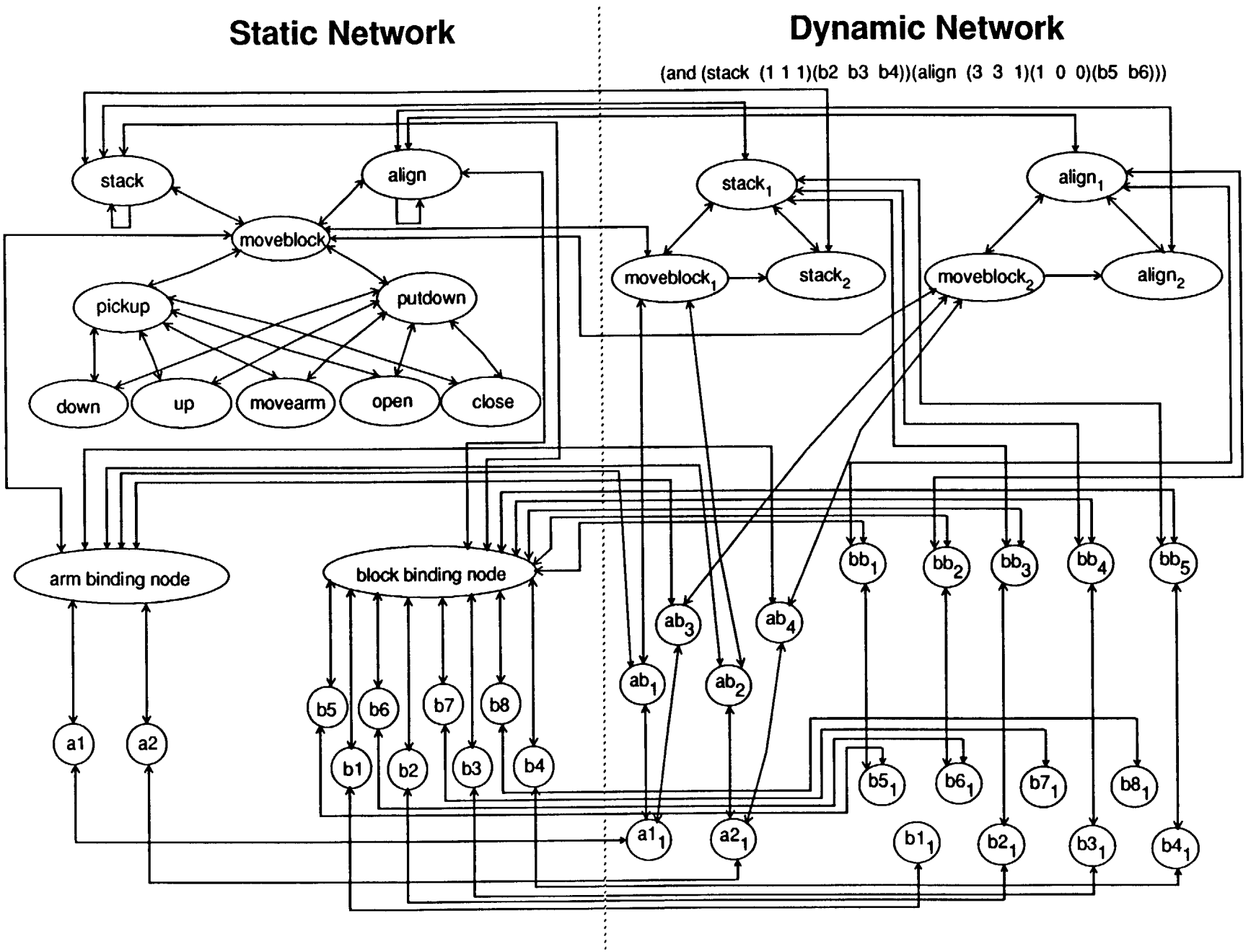


Figure 11: Creation of the First Level of Child Actions

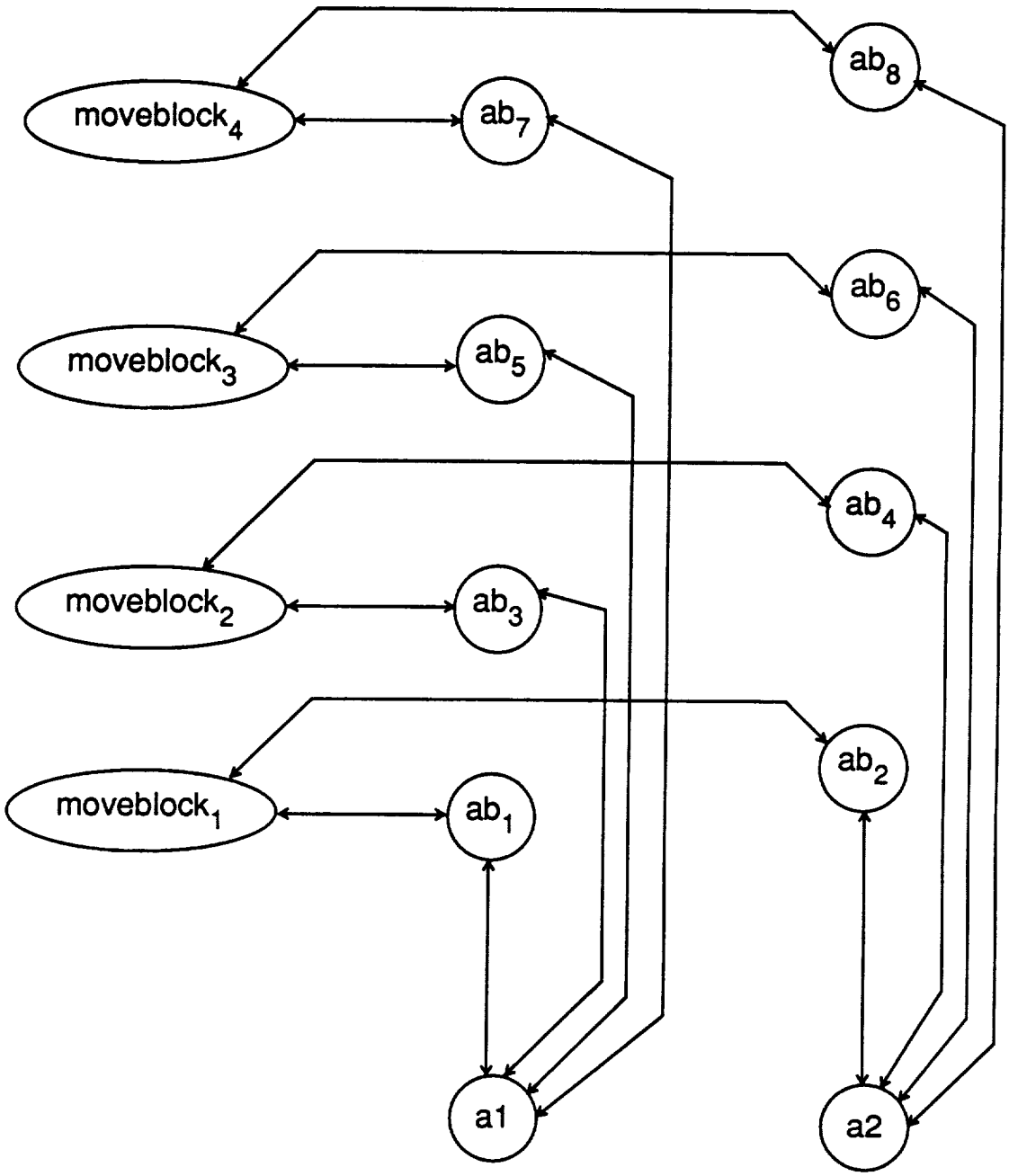


Figure 12: Variable Binding

Some of the child actions of pickup or putdown action nodes can use an arm other than the one used by the parent action node. If when one of these child action nodes is created there are no arms available for which the new child can compete, the new child node inherits the arm binding from its parent pickup or putdown node. This is more efficient than having the child node wait until an arm node becomes available and then spend time competing for it.

Distributed Plan Generation and Execution

Plan generation and execution occur simultaneously in the AutoPlan system parallel processing network. While some nodes may be growing their child nodes (plan generation) other nodes in the network may be actively competing for resources or executing their particular action (plan execution). There are not two distinct plan generation and execution phases of processing.

The plan generation and execution process is really the emergent behavior of the execution of a number of functions by each node in a parallel processing network. On a per-node basis planning can be thought of as the *method* of parallel processing used by each node in the network. The methods for various sets of nodes in the network differ. Typically, a single method is used for each node in a set. For example, each arm node uses the same method. The exception to this is action nodes. Each action node has its own method. However, for the sake of discussion, a generic action node method is presented.

A method for action nodes is responsible for:

- competing for resources (sending and receiving output to/from binding nodes)
- sending activation out to neighboring nodes (lateral action nodes, child action nodes, and binding nodes)
- receiving input from neighboring nodes (parent, child, and lateral action nodes, binding nodes)
- calculating the new activation value for the node based on the input from neighboring nodes. (A node cannot become fully active unless all nodes preceding it in sequence have finished executing and, if it is a non-terminal node, until all its children have also finished executing.)
- generating messages to create child action nodes and their corresponding binding nodes (if necessary) once all the variable bindings for this parent action node are complete
- letting the parent action node and subsequent action nodes in sequence know when this node has finished executing (e.g., when its activation value reaches 1.0) or if it has failed. If at least one child node fails to complete executing then the parent's execution is considered failed as well.

The method for bindings nodes is responsible for:

- sending output to action nodes and resource nodes (e.g., arm, block, or grid location)

- receiving input from action nodes and resource nodes
- updating the activation values of the binding node
- recognizing when multiple action nodes are trying to access a single resource
- communicating to an action node the resource to which that node is bound when the activation value of this node is 1.0
- taking the resource to which it is connected out of the competition once this node's activity reaches 1.0

The methods for arm, block, and grid location nodes simply:

- send output to any binding node competing for it
- receive input from any bindings node competing for it
- update the activation value based on input received and output sent from/to neighboring nodes (binding nodes)

As illustrated above, the method for a node is responsible for all internode communication. The information communicated is a combination of numeric and symbolic. Numeric activation values are used for competition for resources and for activating action nodes. Uses of symbolic information include the binding and unbinding of variables, and the communication of the execution status of an action node.

5.0 PLAN MONITOR AND DIAGNOSIS

Anomalous conditions (manifestations) are detected by the plan monitor either when differences are observed between the ideal state and the real state or when plan actions fail due to unexpected conditions. Based on the manifestations present, the monitor performs diagnosis to determine a cause(s) for the existing manifestations. Causal associations between manifestations and causes are used by the diagnosis program. The monitor then communicates the final diagnosis to the planner so that it can react and replan accordingly. This information consists of messages saying that a resource in the blocks world (e.g., block, arm, grid location) is unavailable (either on a temporary or short-term basis) or that a resource has a different state than its current state in the ideal world.

Plan generation and execution can continue while diagnosis is being done. The monitor sends preliminary information about observed manifestations to the planner before diagnosis is performed so that the part of the plan that is affected by the manifestations can pause its execution while diagnosis takes place. Meanwhile other unaffected parts of the plan can continue to be generated and executed.

Manifestations

A manifestation is a phenomenon that serves as an indication of a disorder. A manifestation may indicate that one or more disorders are present in a system. In the extended blocks world, a manifestation occurs when any state in the real world does not match the

corresponding state in the ideal world maintained by the planner or when the execution of an action fails due to some unexpected occurrence. For example, if in the ideal world block b1 is at location $\vec{loc}_c$ but in the real world block b2 is at location $\vec{loc}_d$ then the difference between the ideal state and the real state indicates that a manifestation is present. Block b2 may be at a location other than that expected by the planner ($\vec{loc}_c$), for many reasons. For example, the cause of the manifestation may be that an arm dropped the block at a different location than expected ($\vec{loc}_d$) due to some problem with the arm or the cause may be that a person bumped the table and therefore moved the block to a different location ($\vec{loc}_d$) after it had been placed in the anticipated location ($\vec{loc}_c$) by an arm. Either one of these scenarios could have caused the observed manifestation. It may also be the case that more than one disorder may combine to cause a single manifestation.

Manifestations in the Extended Blocks World	
Manifestation	**Description**
displcd_block(b)	block b is displaced; it is in a location other than expected
miss_block(b)	block b is missing; its location is currently unknown
displcd_arm(a)	arm a is displaced; it is in a location other than expected
Varm_stop(a)	arm a is stopped in the vertical direction; in other words something is preventing arm a from moving either up, down, or in both directions.
nopickup(a, b)	for some unknown reason, arm a did not pick up block b

Causes (Disorders)

Manifestations appear as a result of causes or disorders. The following table lists the causes being modelled in the extended blocks world.

Causes of Manifestations in the Extended Blocks World	
Cause	**Description**
fall_stack($\vec{loc}$)	stack at $\vec{loc}$ fell
slip_block(a, b)	block b slipped from arm a
bump_block(b)	block b bumped out of place
excess_hand_press(a)	the hand of arm a exerted excess pressure
hand_stuck_open(a)	the hand of arm a is stuck open
Hmotor_imp(a)	the motor of arm a which controls horizontal movement is impaired
Vmotor_imp(a)	the motor of arm a which controls vertical movement is impaired

Causal Associations

The diagnostic problem solving method used by the monitor to do diagnosis uses causal associations between manifestations and disorders [Peng and Reggia, 1989]. Each causal link between a disorder and manifestation is associated with a probability representing the strength of the causal association. The causal associations for the manifestations and disorders modelled for the extended blocks world are represented pictorially in Figure 13.

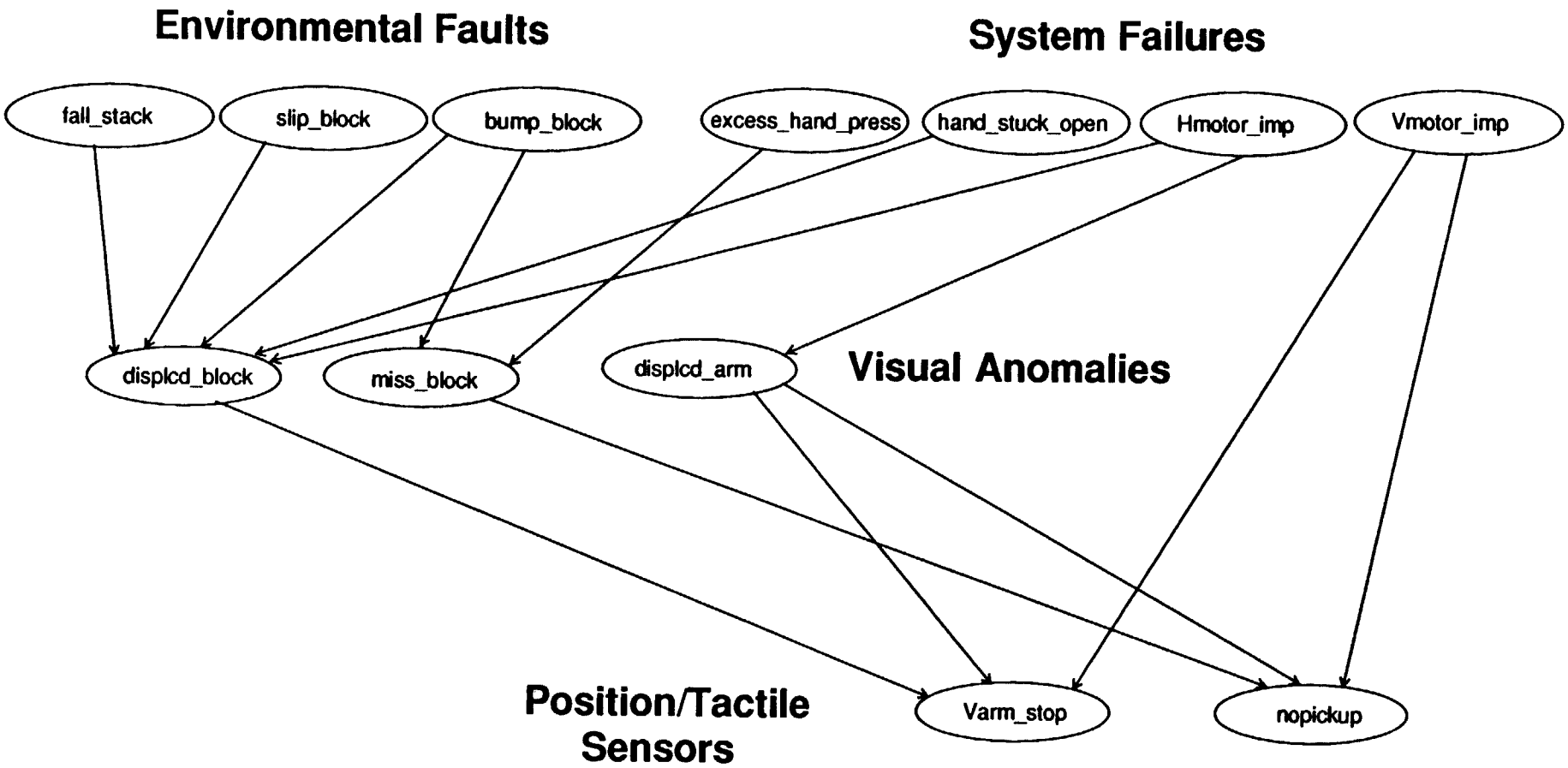


Figure 13: Causal Associations (Static Diagnostic Network)

Information Communicated from Monitor and Diagnosis Component to the Plan Generation and Execution Component

Based on its final diagnosis for a given set of observed manifestations, the monitor communicates to the planner the states of the real world which differ from the states of the ideal world maintained by the planner, and the diagnosis for the observed manifestations. The planner then updates the ideal state of the extended blocks world and replans according to the communicated changes in state and the diagnosis.

6.0 REPLANNING

Replanning occurs when an anomalous condition is detected. Specifically, when a manifestation is detected by the monitor replanning must occur. The replanning operation may be as minor as resetting some attribute values (e.g., a new grid location for a block) and then just continuing planning, or it may require more extensive resetting of attributes, recompetition for limited resources, and changes in the actual plan hierarchy. Replanning is really part of the functionality of the planner but is discussed separately here for the sake of clarity.

Basically replanning must be able to handle two types of situations. One situation occurs when a resource in the extended blocks world is no longer available (e.g., an arm is broken). The second situation occurs when a resource is in a state other than that expected (e.g., a block is at $\vec{loc}_d$ instead of at $\vec{loc}_c$). In either of these two scenarios, the planner must also take into account the diagnosis information sent to it by the monitor before making any replanning decisions.

Replanning When a Resource is No Longer Available

There are two possible situations in which the planning system can be when a particular resource becomes unavailable. One, the planning system is not using that resource at the present time. Two, the resource is being actively used by the planning system.

When a resource which is not being used by the planning system suddenly becomes unavailable the planner simply takes steps to make that resource unavailable. The exact mechanism for this is discussed below. AutoPlan automatically adapts to the loss of the resource since the system only utilizes resources that are part of the self-processing network. The system exhibits a high level of adaptability and fault tolerance.

If a resource is actively being used by the plan when it becomes unavailable replanning is more complicated. The action node to which the resource is bound is notified that the resource is no longer available. If another resource of the same type (e.g., an arm) is available to be competed for then the node competes for that resource. If the action node wins the competition, then all its child action nodes are reexecuted with the new variable binding. If the action node does not win the competition for the resource or there are no resources of the same type available for which to compete, the action node sends a message to the action node one level higher in the action hierarchy. That action node in turn sends a message to any unexecuted child nodes below the current level in the hierarchy instructing them to release any variable bindings of the type required by the action node whose resource became unavailable. If no resources of the required type become free by this process, the highest level action node to receive the message sends the message one level higher in the hierarchy.

This process continues until the top of an action hierarchy is reached or until a resource of the required type has been freed. For example, in a stack hierarchy multiple arms may be bound to multiple moveblock operations. If an arm for one of the moveblock operations fails and no other arms are available this replanning process eventually causes the release of at least one of the arms bound to one of the other moveblock operations later in sequence in the same action hierarchy(e.g., the stack action hierarchy). Descendent nodes recompete for the disabled type of resource. The action node who lost its resource will win the competition and execution will continue at that point in the hierarchy with the new variable binding. Other variable bindings stay intact. If the message is passed all the way to the top of the action hierarchy and no resource of the type required is freed, then the action node must wait and compete for a resource currently being used by some other action hierarchy (e.g., the align action hierarchy).

When a resource becomes unavailable, it must delete itself from both the static and dynamic plan hierarchies. A plan fails when a resource (e.g., an arm) becomes unavailable and there are no other resources of the same type left in the static network. This condition will be detected by the static binding nodes. If a static arm binding node has no outgoing connections to any static arm nodes then it must generate a message indicating that the plan is a failure because there are no more arms available.

Example

Using the example illustrated in Figures 4 through 11 one can examine what will happen when arm 1 (node $a1_1$) becomes unavailable.

When the monitor detects that arm 1 is displaced, it sends a message to the dynamic instance of arm 1 (node $a1_1$) telling it to pause its execution. Node $a1_1$ then relays this message to the dynamic arm binding node to which it is connected, node ab_1. Node ab_1 then relays this information to the action node $moveblock_1$. Node $moveblock_1$ then sends this message to all its descendent action nodes. If a descendent node is not a primitive action then it passes the message along to its descendents. If it is a primitive action then it stops its execution.

While the plan execution is paused the diagnostic component determines that the horizontal motor of the arm is impaired. It then sends a message to this effect to the dynamic arm node $a1_1$. The arm node knows that it cannot recover from such an impairment so it must delete itself from the system.

Before it deletes itself, the static arm node a1 sends a message to the static arm binding node that it is deleting itself. Upon receipt of this message the static arm binding node checks to make sure that it still connects to at least one other static arm node. If it will not have at least one other connection after the static arm a1 is deleted the static arm binding node will generate a message indicating that the plan is a failure because there are no more arms available.

The dynamic arm binding node sends a message to the dynamic action node $moveblock_1$ telling it that it is deleting itself and that arm 1 is no longer available. The $moveblock_1$ node must then set its arm variable to be unbound. It will have to recompete for another arm. Since in this example there are only two arms available, the $moveblock_1$ node will have to wait until the $moveblock_2$ node is finished with the second arm at which point the $moveblock_1$ node will compete with any other action nodes which want to make use of the arm.

If originally there were three arms available and the third arm was bound to a $moveblock_3$ node, a child of the $stack_2$ node, then the $moveblock_1$ node would have to use

the message passing process described above to get the $moveblock_3$ node to relinquish its binding of the third arm so that the $moveblock_1$ action could compete for the third arm.

Replanning When a Resource is in a Different State Than Expected

The tables in the previous section on the plan monitor describe differences in the state of the ideal world and the state of the real world which the replanning system accomodates. When any of these differences in state occur, replanning takes place. As is the case when a resource becomes unavailable there are two possible situations in which the planning system can be when the state of a particular resource changes. One, the planning system is not using that resource at the present time. Two, the resource is being actively used by the planning system. Depending on the diagnosis for the observed state difference, if the resource whose state is different than the expected state is not currently bound to any action node then the planner updates the state of that resource to reflect its current state in the real world. If the resource is currently being used by an action node in the planning system then more extensive replanning is done.

The procedure for handling a difference in the state of a resource currently in use by the planner is similar to the procedure undertaken when a resource becomes unavailable. The action node to which the resource is bound is notified that the resource has a different state than expected. The state of the resource is then updated to reflect its current state in the real world. Sometimes multiple resources may be involved; the same procedure is undertaken for each resource having a different state. For example, if in the ideal world the location of a block is $\vec{loc}_c$ and in the real world the hand of arm a1 is holding the block, then the real states of the arm and the block are in conflict with the states of the block and arm in the ideal world.

Once the state of the ideal world is updated to match the state of the real world, the planner attempts to continue planning starting with the nodes to which the resources are currently bound. If this is not possible the planner then tries moving up one level in the plan hierarchy and executing the plan from there. The replanning procedure differs slightly depending on the type of resource involved, the nature of the change in state, and the diagnosis for the difference in state.

7.0 MIRRORS/II IMPLEMENTATION OF THE SYSTEM COMPONENTS

The Plan and the Ideal State

The plan is represented by the parallel processing network created by MIRRORS/II and the planner. In MIRRORS/II, a self-processing network has two parts: a static network which basically defines legal node connections and a dynamic network composed of instantiations of various parts of the static network. MIRRORS/II creates the static network based on user specifications. It also creates the dynamic arm, block, and grid location nodes based on the static network. The planner generates the dynamic action and binding nodes of the network based on the goals posted to the system and the current state of the network. The ideal state of the system is represented by the attributes of the nodes in the dynamic network. The sequence of actions taken to achieve the given goals is represented by the order in which the action nodes in the plan become active.

Plan Generation and Execution

The process of plan generation and execution is an emergent behavior of the self-processing network. Each node has its own method of spreading numeric activation levels and symbolic information. It is the parallel execution by each node in the network of its local method that creates this emergent global planning behavior. The details of each method were enumerated earlier in Section 4.0.

Real State

It is not yet clear how the real state will be implemented in the MIRRORS/II environment. It is highly probable that it will be implemented as an event which communicates real state information or manifestations to the plan monitor.

Plan Monitor and Diagnosis

Implementation details for the plan monitor are also still being worked out. Most likely it will be an event that is executed once every iteration.

8.0 DISCUSSION

AutoPlan is a complete planning system which includes plan generation, execution, monitoring, and replanning and is based entirely on local operations. It operates in an extended blocks world which is rich enough to provide many challenging features that would be encountered in a real planning and control environment: multiple simultaneous goals, parallel as well as sequential action execution, action sequencing determined not only by goals and their interactions but also by limited resources (e.g., three tasks, two acting agents), the need to interpret unanticipated events and react appropriately through replanning, etc. Conflicts over limited resources are resolved using competitive activation techniques.

The long-term goal of this work is to formulate robust computationally-effective information processing methods for the distributed control of semiautonomous exploration systems, e.g., the Mars Rover. While AutoPlan takes a significant step in this direction it is admittedly limited in the face of the complexity of semiautonomous systems like the Mars Rover. The Rover will operate in a significantly more complex world than the extended blocks world: there is a lot of information about Mars which is unknown, the terrain is highly variable, the potential for mechanical failure is much higher, communication to Earth takes a long time and is subject to corruption, etc. The current design of the Rover includes separate functional modules (e.g., navigation module, science module, etc.) which do some planning independently but also interact with a central coordinator [Johnston, 1989]. Some responsibilities of the coordinator include integrating module plans into a master plan and coordinating resource allocation. It is not clear how a system such as AutoPlan fits into such a scenario. Perhaps separate AutoPlan systems could be used for each module as well as for the coordinator. Many details remain to be worked out before a simulation of a representative subset of the activities of a semiautonomous system like the Mars Rover can be attempted.

Although the AutoPlan system has its limitations it still makes many contributions to planning research. AutoPlan does reactive planning, resource-limited planning, conditional planning, deferred planning, and hierarchical planning, all of which are implemented using strictly local operations. The AutoPlan system has an integrated plan generation and execution process which allows it to react quickly to unexpected changes in the environment. The plans generated by the planner are hierarchical in nature. Conflicts over limited resources in the extended blocks world are resolved using competitive activation methods. Conditional

planning is implemented via conditional links in the static plan hierarchy. Certain dynamic action nodes are only generated under specific environmental conditions. Many planning decisions (e.g., which resource an action uses or which child nodes to generate) are deferred as long as possible.

9.0 REFERENCES

Bourret, P., Goodall, S., and Samuelides, M. Optimal Scheduling by Competitive Activation: Application to the Satellite-Antenna Scheduling Problem. *Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, June 1989, 565-572.

Bourret, P., Remy, F., and Goodall, S. A Special Purpose Neural Network for Scheduling Satellite Broadcasting Times. *Proceedings of the International Joint Conference on Neural Networks*, II, Jan. 1990, 535-538.

Cheeseman, P. Planning and Scheduling in AI, in Heer, E. and Lum, H. (Eds.), *Machine Intelligence and Autonomy for Aerospace Systems*, American Institute of Aeronautics and Astronautics, Inc., Washington, D.C., 1988.

D'Autrechy, C.L., Reggia, J.A., Sutton, G.G. III, Goodall, S.M., Tagamets, M., and Marsland, P. *MIRRORS/II Reference Manual*, UMIACS-TR-88-41 and CS-TR-2043, Department of Computer Science, University of Maryland, 1988.

D'Autrechy, C.L. and Reggia, J.A. Parallel Plan Execution with Self-Processing Networks, *Telematics and Informatics*, 6, 1989, 145-157.

D'Autrechy, C.L. and Reggia, J.A. *An Overview of Sequence Processing by Connectionist Models.* UMIACS-TR-89-82 and CS-TR-2301, Department of Computer Science, University of Maryland, 1989.

Goodall, S. and Reggia, J. Competitive Activation Methods for Dynamic Control Problems. *Proceedings of the International Joint Conference on Neural Networks*, II, Jan. 1990, 343-346.

Hendler, J. *Integrating Marker-Passing and Problem-Solving.* Lawrence Erlbaum, 1988.

Hendler, J. Marker-Passing Over Microfeatures: Towards a Hybrid Symbolic/Connectionist Model, *Cognitive Science*, 13, 1989, 76-106.

Johnston, M. *Pathfinder Planetary Rover Design Reference Mission*, NASA/Ames Research Center, AI Research Branch, UPN 591-11-41, September 1989.

Jordan, M. Attractor Dynamics and Parallelism in a Connectionist Sequential Machine. *Proceedings of the Eight Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, 531-546, 1986.

Maes, P. How to Do the Right Thing, *Connection Science*, 1, 1989, 291-323.

Miyata, Y. An Unsupervised PDP Learning Model for Action Planning. *Program of the Tenth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum, 223-229.

Peng, Y. and Reggia, J. A Connectionist Model for Diagnostic Problem Solving, *IEEE Transactions on Systems, Man, and Cybernetics*, 19, 1989, 285-298.

Reggia, J., Methods for Deriving Competitive Activation Mechanisms, *Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, June 1989, 357-363.

Reggia, J. and Sutton, G.G. III, Self-Processing Networks and Their Biomedical Implications, *Proceedings of the IEEE*, 76, 1988, 680-692.

Sliwa, N. and Soloway, D. A Lattice Controller for Telerobotic Systems. Presented at the 1987 American Controls Conference.

Sliwa, N. Behavioral Networks as a Model for Intelligent Agents. Presented at the 1989 Space Operations Automation and Robotics (SOAR) Conference, NASA Johnson Space Center, Houston, Texas, 1989.

Vere, S. Planning in time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 3, 1988, 246-267.*

Whitehead, S. and Ballard, D., Connectionist Designs on Planning, Proceedings of the 1988 Connectionist Models Summer School, D. Touretzky, G. Hinton, and T. Sejnowski (Eds.). Morgan Kaufman, 357-370.

Whitfield, K., Goodall, S., and Reggia, J. A Connectionist Model for Dynamic Control, *Telematics and Informatics*, 6, 1989, 375-390.

Appendix A
Plan Generation and Execution Details

Assumptions

1. A robot arm can only hold one block at a time.
2. Using a 3-dimensional notation.
3. The arguments are always valid.

    - *a* always evaluates to an existing robot arm in our environment
    - *b, c* always evaluate to blocks in our blocks world
    - *blocks* is a list of blocks (e.g., $b_i, \ldots, b_j$)
    - $\vec{loc}$ is a 3-D vector representing a grid location
    - $\vec{d}$ is a 3-D vector in which one of the components is ± 1 and the rest of the components are zero (e.g., [0 0 -1]).

4. Each block knows about the six positions adjacent to it. If the block is at position (x, y, z), the six adjacent positions are:

    (x-1, y, z)
    (x+1, y, z)
    (x, y-1, z)
    (x, y+1, z)
    (x, y, z-1)
    (x, y, z+1)

5. The table is at z = 1. z < 1 is an illegal z value.
6. The robot arm moves in a plane (z = ZMAX) above the highest stack of blocks.
7. The robot arm moves horizontally only when the arm is in the highest z-plane (e.g., from (x_{source}, y_{source}, ZMAX) to (x_{dest}, y_{dest}, ZMAX). The arm moves vertically along a line segment between (x, y, z=1) to (x, y, z). The arm cannot move both horizontally and vertically at the same time.
8. The bounds on x, y, and z are [XMIN, XMAX], [YMIN, YMAX], and [1, ZMAX] respectively.
9. The highest stack can only go to a height of ZMAX-1 because the arm needs to move at height ZMAX.

Notations

1. Subscript meanings:

    a = some arm
    d = destination
    i = some block
    j = some block
    r = random
    s = source

2. ($\vec{loc}_i$) is the location of block b_i.
3. ($\vec{loc}_{ri}$) where *r* stands for random, is the random location of block b_i.
4. If $\vec{loc}$ = [x y z] then $\vec{maxloc}$ = [x y ZMAX].

Possible States of the System

Much of the state of the extended blocks world can be described using the predicates listed in the table below.

Possible States of the System	
State	**Description**
openhand(*a*)	the hand of robot arm *a* is open
empty(*a*)	the hand of robot arm *a* is empty
holding(*a*, *b*)	the hand of robot arm *a* is holding block *b*
loc(*object*, $\vec{loc}$)	the *object*, either an arm or a block, is at location $\vec{loc}$

Possible Actions of the System

The possible actions of the system are described in the following table. The Level column is used to indicate the level of the given action in the overall action hierarchy. Primitive (lowest level) actions are listed first in the table. Following them, actions are listed in order, by level, with the top level actions coming at the end of the table.

Possible Actions of the System		
Action	**Description**	**Level**
down(*a*, $\vec{loc}$)	move robot arm *a* down to location $\vec{loc}$; the z coordinate changes while x and y remain the same	primitive
up(*a*, $\vec{loc}$)	move robot arm *a* up to location $\vec{loc}$; the z coordinate changes while x and y remain the same	primitive
open(*a*)	open the hand of robot arm *a*	primitive
close(*a*)	close the hand of robot arm *a*	primitive
movearm(*a*, $\vec{loc}$)	move robot arm *a* to location $\vec{loc}$; the x and y coordinates change while z remains the same (at ZMAX)	primitive
pickup(*a*, *b*)	pick up block *b* with arm *a*	third
putdown(*a*, *b*, $\vec{loc}$)	put block *b* held by arm *a* down at location $\vec{loc}$	third
moveblock(*a*, *b*, $\vec{loc}$)	using arm *a*, move block *b* to location $\vec{loc}$	second
stack($\vec{loc}$, *blocks*)	make a vertical stack of blocks at location $\vec{loc}$ where the first block in the list is on the top of the stack and the last block in the list is on the bottom of the stack	top
unstack(*blocks*)	take each block starting with the top block off the stack and put it on the table at an unoccupied location	top
align($\vec{loc}$, $\vec{d}$, *blocks*)	make a horizontal stack of blocks where the last block in the list is on the table at location $\vec{loc}$, the next to the last block in the list is on the table at location $\vec{loc}+\vec{d}$, etc.	top
unalign($\vec{d}$, *blocks*)	take each block starting with the first block in the list away from the block it is currently next to and put it at an unoccupied location on the table	top

Connectionist Network

Grid Set

Each grid position is a node which connects to its six neighbors within a radius of one and to a grid binding node.

Grid Set Attributes		
Name	**Value**	**Description**
block	dynamic block node or nil	block which occupies that square of the grid
arm	dynamic arm node or nil	arm which is positioned at that square of the grid
maxloc	dynamic grid node	grid node which has the same x,y coordinates as the grid node itself but has ZMAX as its z coordinate

These attribute values change dynamically.

Blocks Set

Each node represents a single block. A node has a connection to a block binding node.

Blocks Set Attributes		
Name	**Value**	**Description**
location	dynamic grid node	The block is at location $\vec{loc}$ in the three-dimensional grid. The block is either on another block ($z(\vec{loc}) > 1$) or is on the table ($z(\vec{loc}) = 1$).
	dynamic arm node	The block is being held by the hand of the arm represented by the arm node. The block is at the same location as the arm.

NOTE: The two possible values for the location attribute are mutually exclusive.

Arm Set

Each node in this set represents one robot arm in the blocks world environment. A node has connections to arm binding nodes.

Arm Set Attributes		
Name	**Value**	**Description**
location	dynamic grid node	The arm is currently positioned at location $\vec{loc}$.
holding	dynamic block node or nil	The block which is currently being held by the arm.
handopen	t or nil	Boolean value indicating whether the robot hand is open (t) or closed (nil).

Arm Bindings Set

Each node in this set connects to one action node and one arm node. These nodes are used by action nodes to compete for arm nodes. When an arm binding node becomes fully active, it takes steps to bind the action node to the arm node to which it is connected.

Block Bindings Set

Each node in this set connects to one action node and one block node. These nodes are used by action nodes to compete for block nodes. When a block binding node becomes fully active, it takes steps to bind the action node to the block node to which it is connected.

Grid Location Bindings Set

Each node in this set connects to one action node and one grid location node. These nodes are used by action nodes to compete for grid location nodes. When a grid location binding node becomes fully active, it takes steps to bind the action node to the grid location node to which it is connected.

Actions Set

Each node in this set represents an action in a hierarchical plan. There may be more than one dynamic instance of each action node.

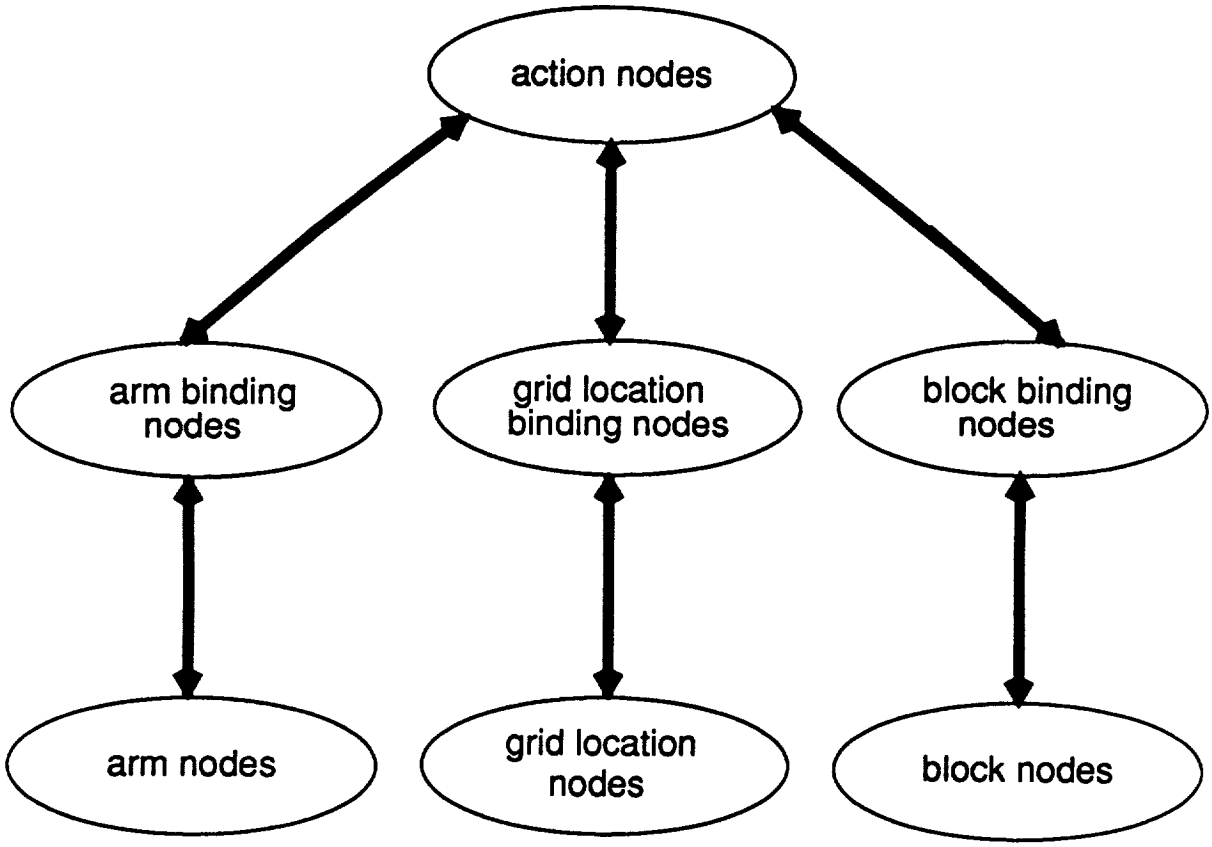
Actions Set Attributes		
Name	**Value**	**Description**
parent	dynamic action node or nil	the action node which is one level above this action node in the action node hierarchy
arm	dynamic arm node or nil	the value of the arm variable for this action node; may be bound at run time or may be inherited from the parent action node; may be nil if action does not have an arm variable (e.g., stack)
blocks	list of dynamic block nodes, dynamic block node, or nil	the value of the b or blocks variable for this action node; often inherited from the parent action node; may be nil if action does not have a b or blocks variable (e.g., up)
location	dynamic grid node or nil	the value of the location variable for this action node; may be nil if action does not have a location variable (e.g., open)
direction	3-D vector in form of [±1 0 0], [0 ±1 0], or [0 0 ±1], or nil	the vector is used to change the current location by adding this vector to the location vector; only one of the components of this vector may be ±1 at a time (e.g., [+1 0 0]); may be nil if action does not have a direction variable (e.g., stack)
vars	list of atoms (variable names)	the list of variables for this action node
bindings	list of lists of variable bindings	describes the variable bindings for each child node of this action node

Actions Set Attributes (continued)		
Name	**Value**	**Description**
lat-seq-conns	list of lists of action node names	describes which child nodes of this action node connect laterally to which other child nodes of this action node. These lateral connections provide sequencing information in terms of which nodes can be executed in parallel and which must be executed in sequence.
effects	list of state changes or nil	describes the effects, in terms of state changes, which are the result of executing a primitive action; nil for non-primitive actions

Value of the Effects Attribute of the Primitive Action Nodes

Value of the Effects Attribute		
Action Node	**Effects Value**	**Explanation**
down	(setf (loc a)(loc self))	loc of arm is loc argument to action
up	(setf (loc a)(loc self))	loc of arm is loc argument to action
open	(setf (handopen a) t) (cond ((holding a) (setf (blockat (loc a))(holding a)) (setf (loc block)(loc a)) (setf (holding a) nil)))	hand is now open if the hand was holding a block grid loc now occupied by block loc of block is grid loc arm no longer holds block
close	(setf (handopen a) nil) (cond ((eq (parent self) 'pickup) (setf (holding a)(blockat (location a))) (setf (loc (blockat (loc a))) a) (setf (blockat (loc a)) nil)))	hand is now closed if arm is picking up block arm now holds block at its location block is now held by arm block held by arm is no longer at grid loc
movearm	(setf (loc a)(loc self))	loc of arm is loc argument to action

Connectionist Network Architecture



Functions Used to Access Slots in Node Data Structures

These functions are primarily used to determine if action preconditions have been satisfied and to bind variables to values "inherited" from parent nodes.

Access Functions	
Function	**Description**
arm(*action_node*)	Returns the value of the arm variable (attribute) of the specified *action_node*. The value can be either an arm node or nil. If the value is nil then the arm variable (attribute) needs to be bound at the current level.
block(*action_node*)	Returns the value of the blocks attribute of the specified *action_node*. The value can be either a list of dynamic block nodes, a dynamic block node, or nil.
blockat(*grid_node*)	Returns the value of the block attribute of the specified *grid_node*. The value can be either a dynamic block node or nil.
dir(*action_node*)	Returns the value of the direction attribute of the specified *action_node*. The value can be a direction vector or nil.
holding(*arm_node*)	Returns the value of the holding attribute of *arm_node*. The value can be either a dynamic block node or nil.
loc(*action_node*)	Returns the value of the location attribute of the specified *action_node*. The value is a vector in the form of [x y z].
loc(*arm_node*)	Returns the value of the location attribute of the specified *arm_node*. The value is a vector in the form of [x y z].
loc(*block_node*)	Returns the value of the location attribute of the specified *block_node*. The value is a vector in the form of [x y z].
maxloc($\vec{loc}$)	Assuming $\vec{loc}$ is [x y z], then this function returns the dynamic grid node which represents vector [x y ZMAX]. Vector [x y ZMAX] is the location in the arm movement plane which corresponds to $\vec{loc}$.
neighbor($\vec{loc}$, $\vec{dir}$)	$\vec{dir}$ is in the form of [±1 0 0], [0 ±1 0], or [0 0 ±1]. A new vector is obtained by adding $\vec{loc}$ + $\vec{dir}$. The grid node corresponding to the new vector is returned.
parent(*action_node*)	Returns the value of the parent attribute of the specified *action_node*. The value can either be a dynamic action node or nil.
random_grid_loc(*type*)	*type* = 'occupied or 'unoccupied. According to the value specified for *type*, this function returns a pointer to either an occupied or unoccupied location (on the table?).
z($\vec{loc}$)	Returns the value of the z coordinate of the $\vec{loc}$ vector.

The following table describes functions which are used to manipulate lists of dynamic block nodes.

Block List Manipulation Functions	
Function	**Description**
firstblock(*block_list*)	Returns the first dynamic block node on the list. This is analogous to (car block_list) in Lisp.
lastblock(*block_list*)	Returns the last dynamic block node on the list. This is analogous to (car (last block_list)) in Lisp.
allbutfirstblock(*block_list*)	Returns a list of dynamic block nodes starting with the second block in the list and ending with the last block in the list. This is analogous to (cdr block_list) in Lisp.
allbutlastblock(*block_list*)	Returns a list of dynamic block nodes starting with the first block in the list and ending with the next to last block in the list.

Preconditions and Effects of Actions/Goals

If a precondition remains true after the goal is accomplished then it is not listed as an effect. Each action is responsible for making sure it has not already been achieved before continuing to check to see if the rest of its preconditions have been achieved. The effects of non-primitive actions are really just the cumulative effects of many primitive actions.

Preconditions and Effects		
Preconditions	**Goal**	**Effects**
$z(\vec{loc}_{current}) > z(\vec{loc})$	down(a, $\vec{loc}$)	loc(a) = $\vec{loc}$
$z(\vec{loc}_{current}) < z(\vec{loc})$	up(a, $\vec{loc}$)	loc(a) = $\vec{loc}$
¬openhand(a)	open(a)	openhand(a) if holding(a) then blockat(loc(a)) = holding(a) loc(holding(a) = loc(a) ¬holding(a)
openhand(a)	close(a)	¬openhand(a) if parent(self) = pickup then holding(a) = blockat(loc(a)) loc(blockat(a)) = a blockat(loc(a)) = nil
$\vec{loc}$ ≠ loc(a)	movearm(a, $\vec{loc}$)	loc(a) = $\vec{loc}$
¬holding(a, b) ¬blockat(neighbor($\vec{loc}$, [0 0 1])) ¬holding(a, c) $max\vec{loc}_b$ = loc(a)	pickup(a, b)	holding(a, b)
b ≠ blockat($\vec{loc}$) ¬blockat($\vec{loc}$) blockat(neighbor($\vec{loc}$, [0 0 -1])) holding(a, b) $max\vec{loc}$ = loc(a)	putdown(a, b, $\vec{loc}$)	loc(b) = $\vec{loc}$ handempty(a)
b ≠ blockat($\vec{loc}$)	moveblock(a, b, $\vec{loc}$)	blockat($\vec{loc}$) = b
b_j ≠ blockat($\vec{loc}$) . . . b_i ≠ blockat(neighbor($\vec{loc}$, [0 0 n-1]))	stack($\vec{loc}$, *blocks*) n = the number of blocks	blockat($\vec{loc}$) = b_j . . . blockat(neighbor($\vec{loc}$, [0 0 n-1])) = b_i
b_j = blockat($\vec{loc}$) . . . b_i = blockat(neighbor($\vec{loc}$, [0 0 n-1]))	unstack($\vec{loc}$, *blocks*) n = the number of blocks	blockat($\vec{loc}_{rj}$) = b_j . . . blockat($\vec{loc}_{ri}$) = b_i
b_j ≠ blockat($\vec{loc}$) . . . b_i ≠ blockat(neighbor($\vec{loc}$, $\vec{d}$ * n-1)))	align($\vec{loc}$, $\vec{d}$, *blocks*) n = the number of blocks	blockat($\vec{loc}$) = b_j . . . blockat(neighbor($\vec{loc}$, $\vec{d}$ * n-1)) = b_i
b_j = blockat($\vec{loc}$) . . . b_i = blockat(neighbor($\vec{loc}$, $\vec{d}$ * n-1)))	unalign($\vec{d}$, *blocks*) n = the number of blocks	blockat($\vec{loc}_{rj}$)) = b_j . . . blockat($\vec{loc}_{ri}$) = b_i

Variable Bindings

Bindings for a descendent action may vary according to the identity of the parent and the condition (if any) on the link from the parent to the descendent. So, how should we store this binding information which is dependent on the parent action? Since the parent stores the lateral connection information which is parent-dependent, perhaps the parent should also store the variable binding information which is parent dependent.

Descendent Variable Bindings

pickup(a,b)

moveblock(a', c', $\vec{loc}_r$)

a' = ***To Be Bound*** or (arm (parent self))
c' = (blockat (neighbor (loc (blocks (parent self))) [0 0 +1]))
$\vec{loc}_r$ = (random_grid_loc 'unoccupied)

putdown(a, c, $\vec{loc}_r$)

a = (arm parent)
c = (holding (arm (parent self)))
$\vec{loc}_r$ = (random_grid_loc 'unoccupied)

up(a, $\vec{maxloc}_a$)

a = (arm (parent self))
$\vec{maxloc}_a$ = (maxloc (loc (arm (parent self))))

movearm(a, $\vec{maxloc}_b$)

a = (arm (parent self))
$\vec{maxloc}_b$ = (maxloc (loc (blocks (parent self))))

open(a)

a = (arm (parent self))

down(a, $\vec{loc}_b$)

a = (arm (parent self))
$\vec{loc}_b$ = (loc (blocks (parent self)))

close(a)

a = (arm (parent self))

up(a, $\vec{maxloc}_a$)

a = (arm (parent self))
$\vec{maxloc}_a$ = (maxloc (loc (arm (parent self))))

putdown(a, b, $\vec{loc}$)

moveblock(*a'*, *c*, $\vec{loc}_r$)

a' = ***To Be Bound*** or (arm (parent self))
c = (blockat (loc (parent self)))
$\vec{loc}_r$ = (random_grid_loc 'unoccupied)

moveblock(*a''*, *c'*, $\vec{loc}$)

a'' = ***To Be Bound*** or (arm (parent self))
c' = (blockat (random_grid_loc 'occupied)
$\vec{loc}$ = (neighbor (loc (parent self)) [0 0 -1])

pickup(*a*, *b*)

a = (arm (parent self))
b = (blocks (parent self))

up(*a*, $\vec{maxloc}_a$)

a = (arm (parent self))
$\vec{maxloc}_a$ = (maxloc (loc (arm (parent self))))

movearm(*a*, $\vec{maxloc}$)

a = (arm (parent self))
$\vec{maxloc}$ = (maxloc (loc (parent self)))

down(*a*, $\vec{loc}$)

a = (arm (parent self))
$\vec{loc}$ = (loc (parent self))

open(*a*)

a = (arm (parent self))

up(a, $\vec{maxloc}$)

a = (arm (parent self))
$\vec{maxloc}$ = (maxloc (loc (parent self)))

close(*a*)

a = (arm (parent self))

moveblock(a, b, $\vec{loc}$)

pickup(*a*, *b*)

a = (arm (parent self))
b = (blocks (parent self))

putdown(*a*, *b*, $\vec{loc}$)

a = (arm (parent self))
b = (blocks (parent self))
$\vec{loc}$ = (loc (parent self))

stack($\vec{loc}$, blocks)

moveblock(*a*, b_j, $\vec{loc}$)

a = ***To Be Bound*** or (arm (parent self))
b_j = (lastblock (blocks (parent self)))
$\vec{loc}$ = (loc (parent self))

stack($\vec{loc}'$, *blocks'*)

$\vec{loc}'$ = (neighbor (loc (parent self)) [0 0 +1])
blocks' = (allbutlastblock (blocks (parent self)))

unstack(blocks)

moveblock(*a*, b_i, $\vec{loc}_{ri}$)

a = ***To Be Bound*** or (arm (parent self))
b_i = (firstblock (blocks (parent self)))
$\vec{loc}_{ri}$ = (random_grid_loc 'unoccupied)

unstack(*blocks'*)

blocks' = (allbutfirstblock (blocks (parent self)))

align($\vec{loc}$, $\vec{d}$, blocks)

moveblock(*a*, b_j, $\vec{loc}$)

a = ***To Be Bound*** or (arm (parent self))
b_j = (lastblock (blocks (parent self)))
$\vec{loc}$ = (loc (parent self))

align($\vec{loc}'$, $\vec{d}$, *blocks'*)

$\vec{loc}'$ = (neighbor (loc (parent self)) (dir (parent self)))
$\vec{d}$ = (dir (parent self))
blocks' = (allbutlastlblock (blocks (parent self)))

unalign($\vec{d}$, blocks)

moveblock(a, b_i, $\vec{loc}_{ri}$)

a = ***To Be Bound*** or (arm (parent self))
b_i = (firstblock (blocks (parent self)))
$\vec{loc}_{ri}$ = (random_grid_location 'unoccupied)

unalign($\vec{d}$, *blocks'*)

$\vec{d}$ = (dir (parent self))
blocks' = (allbutfirstlblock (blocks (parent self)))

Hierarchy Diagrams

The following diagrams show the relationship between each action node and the action nodes one level below it in the hierarchy of action nodes.

Changes to MIRRORS/II

(1) The ability to have a different method for different nodes in a set. This implies that the method statement should also become a legal node statement.

(2) The ability to add/delete a new dynamic node to/from a network. This implies the need for a new constructor.

(3) The ability to add/delete node connections dynamically. This also implies the need for a new constructor.

(4) The ability to have symbolic weights. May sometimes want to have both symbolic and numeric weights on the same link.

(5) The ability to pass symbolic as well as numeric information over links. (We can sort of do this already but it's probably worth rethinking in terms of having links with symbolic weights.)

(6) Two new events — goal and state. The goal event sets the goal which the planning system is trying to achieve. The state event indicates the current (initial) state of the system. It is possible that the state event could also be used to indicate that the state has changed part way through the planning process.

(7) A new connection logic is needed in order to indicate parallelism (e.g., do one or more of the following independent things in parallel).

Appendix B
Plan Monitor and Diagnosis Details

Possible States of Objects in the Planning System

The following table describes predicates used to represent some of the possible states of objects in the planning system. The predicates in this table in combination with the predicates in the table of possible states from Appendix A can be used to describe the possible states of the extended blocks world.

Possible States of the System Pertaining to Plan Monitoring	
State	**Description**
handop(a)	the hand of arm a is operative
armop(a)	the arm of arm a is operative
lost(b)	the location of block b is unknown
arm_busy(a, action)	arm a is in use by action node action
arm_res(a, action)	arm a is reserved by action node action

Differences in State

The following three tables describe differences in state between the ideal world and the real world which the monitor can detect and for which it can diagnose causes. In turn, the replanning part of the planner can replan for each of these possible differences in state.

Possible Differences in the State of a Block	
Ideal State	**Real State**
block is at $\vec{loc}_c$	block is at $\vec{loc}_d$
block held by arm a	block is at $\vec{loc}_c$
block is at $\vec{loc}_c$	block held by arm a

Possible Differences in the State of a Grid Location	
Ideal State	**Real State**
unoccupied	occupied by an arm with an empty, closed hand
unoccupied	occupied by an arm with an empty, open hand
unoccupied	occupied by an arm holding a block
unoccupied	occupied by a block
occupied by an arm with an empty, closed hand	unoccupied
occupied by an arm with an empty, open hand	unoccupied
occupied by an arm holding a block	unoccupied
occupied by a block	unoccupied

In the following table one will not see any states in which a hand is open and is holding a block. The two states, hand open and holding a block, are mutually exclusive.

Possible Differences in the State of an Arm					
Ideal State			Real State		
Location	Holding	Hand	Location	Holding	Hand
$\vec{loc}_c$	no	open	$\vec{loc}_c$	no	closed
$\vec{loc}_c$	no	open	$\vec{loc}_c$	yes	closed
$\vec{loc}_c$	no	closed	$\vec{loc}_c$	no	open
$\vec{loc}_c$	no	closed	$\vec{loc}_c$	yes	closed
$\vec{loc}_c$	yes	closed	$\vec{loc}_c$	no	open
$\vec{loc}_c$	yes	closed	$\vec{loc}_c$	no	closed
$\vec{loc}_c$	no	open	$\vec{loc}_d$	no	open
$\vec{loc}_c$	no	open	$\vec{loc}_d$	no	closed
$\vec{loc}_c$	no	open	$\vec{loc}_d$	yes	closed
$\vec{loc}_c$	no	closed	$\vec{loc}_d$	no	open
$\vec{loc}_c$	no	closed	$\vec{loc}_d$	no	closed
$\vec{loc}_c$	no	closed	$\vec{loc}_d$	yes	closed
$\vec{loc}_c$	yes	closed	$\vec{loc}_d$	no	open
$\vec{loc}_c$	yes	closed	$\vec{loc}_d$	no	closed
$\vec{loc}_c$	yes	closed	$\vec{loc}_d$	yes	closed