A GRAPHICAL PERSPECTIVE ON ROBOT WORKCELL PROGRAMMING

P. Freedman G. Carayannis A. Malowany

McGill Research Centre for Intelligent Machines
3480 University St., Montréal, Québec, Canada H3A 2A7

Abstract

We describe an interactive graphics interface for the programming environment of a robotics workcell consisting of many cooperating elements such robots, sensing systems. To manage the programming complexity, we have developed a graphics interface (GI) to help the user interactively fashion a description of a particular application. The operations of various workcell elements are displayed as nodes, and their inter-dependencies are displayed as arcs. GI then translates the display into a Timed Petri Net representation which is used to generate a time-optimal sequence of operations for each workcell element. This sequence is ultimately downloaded to the workcell runtime system for execution.

Résumé

Il est bien connu que la programmation d'une cellule robotique est un travail complexe. En commençant par les éléments, par exemple, les robots et les systèmes de vision par ordinateur, il faut écrire les programmes de commande spécifiques à chacun afin d'effectuer les tâches requises. Face à ces problèmes, il est apparu nécessaire de concevoir un système de FAO pour la programmation de la cellule. Les résaux de Petri sont particulièrement bien adaptés pour aborder ce genre de problème.

Nous avons développé une interface graphique qui assiste interactivement le processus de spécification d'une application de la cellule. Chaque opération est décrite par (i) les pré-conditions, (ii) les changements d'état de la cellule causés par cette opération, et (iii) la durée d'exécution. Les tâches inutiles ou partiellement décrites sont automatiquement marquées par le système en vue de leur correction. Les opérations sont réprésentées par les nœuds réunis par les arcs qui répésentent les dépendances entre les opérations. Par la suite, tout devient un résau de Petri utilisé pour optimiser la séquence des tâches de chaque élément afin de maximiser le rendement.

1. Introduction

The complexity of modern robotic workcells, consisting of multiple robots, sensing systems, expert systems, etc. makes programming applications enormously difficult. First, functional redundancy among the 'active' workcell elements complicates the decomposition of the global task into workcell operations. Second, the tight coupling among workcell operations due to constraints on precedence and resource sharing makes it difficult to identify possible concurrency. Eventually, a sequential program must be written for each element, and they must then execute concurrently. Graphics can help cope with this complexity, since a graphical *visualization* of the concurrent programs makes evident the distributed nature of the workcell and the interdependencies of the operations. We describe current work on such an interactive graphics interface for our robotic workcell programming environment.

In what follows, the term ***task*** is used to indicate a unit of work, such as "move_object". The term *operation* denotes the assignment of a task to an active workcell element, such as "move_object using robot R1". Finally, we will sometimes speak of the *activities* associated with a task, such as "move_to_object, grasp_object, etc."

This work is part of a larger effort within the Computer Vision and Robotics Laboratory, McGill Research Centre for Intelligent Machines, to develop a sophisticated hierarchical programming environment for a generic robotic workcell.

2. Background

Even when tasks can be pre-assigned to the 'active' workcell elements, a feasible sequence of tasks must still be found. But this is an NP-complete problem[10], which (informally) implies that there exists no efficient (polynomial time) algorithm for obtaining a solution. Therefore, problems of this nature must be solved by searching the space of possible solutions. When the search space is large, a decision support system becomes important. But because a workcell is typically configured to repetitively perform a given application, it is important to determine not just a feasible sequence, but the time-optimal one.

The sequencing problem is often complicated by what we call *internal non-determinism*. This means that at a particular instant in time, alternative operations associated with the same workcell element become enabled, each giving rise to different sequences of workcell events. Thus, although the outcomes of the operations are fixed, there is still non-determinism in the real time behavior of the workcell. The decision support system must therefore explore the consequences of each alternative in order to determine the 'best' one associated with the fastest repetitive sequence.

Our objective is to start with a user description of the application which specifies what must be done i.e. the set of operations, but not the order in which operations occur. The constraints linking the operations can be characterized as (i) those associated with the precedence of workcell operations, and (ii) those associated with the sharing of resources. In order to map both onto a single condition/event formalism, we define a set of *state variables* associated with 'passive' workcell elements such as shared jigs. A state variable might be boolean such as 'jig = { free, occupied }' or multi-valued such as 'board_in_jig = { new, inspected, repaired, checked } . From the state variables and their values, we then define *conditions* which are either true or false eg. [jig = free]. These conditions are used to construct the pre-conditions and state changes of each operation. (The actual selection of state variables is left to the user, since this is closely tied to the application itself; it should be clear that this selection is not unique. This same philosophy is followed in network analysis, where the problem solver is free to choose a set of independent circuits or voltages, or in control systems design, where the choice of state variables is based on engineering judgement.)

To map precedence constraints among operations to pre-conditions, we can define a multi-valued state variable with a distinct value corresponding to each operation. For example, given a repair task and a checking task, the precedence constraint 'repair_task *before* checking_task' could be re-stated as follows. Then a state variable 'board_in_jig' which takes on values { repaired, checked }, can be used to define [board_in_jig = repaired] as the pre-condition of the checking_task.

The mapping of resource constraints to pre-conditions is made simple by creating a distinct state variable for each resource. For example, given a common workarea used by two robots for different tasks, we could include [shared_workarea = free] in the pre-conditions of both robot tasks.

Our intended workcell runtime environment[3] imposes extra structure on our programming paradigm. (i) For each workcell operation, there must be a 1:1 correspondence between the state variables associated with the pre-conditions and the state variables associated with the state changes. (ii) If the same condition is a pre-condition of multiple operations, or if the same condition is a state change of multiple operations, these operations are intended to be mutually exclusive.

Propositional logic seems to provide a natural vocabulary to describe the workcell state, since the truth value of propositions such as [jig = free] change as the operations take place. But propositions are really just *conditions*, and if we think of operations as *events*, then we can draw a correspondence between the theory of condition/event Petri Nets and propositional logic[16].

Petri Nets are gaining increasing importance as a convenient modelling framework for problems in robotics. For example, the 1987 IEEE International Conference on Robotics and Automation devoted two complete sessions to this subject. Unfortunately, there has yet to emerge a standard way of modelling robotic systems, and this makes it difficult to compare research efforts. We decided to begin with the simple formalism of conditions and events, which makes explicit the inter-task constraints. And unlike most of the Petri Net research concerned with 'strategic' factory-wide routing problems i.e. moving workpieces, tool magazines, and other resources between workcells within a Flexible Manufacturing System, we have concentrated on the 'tactical' coordination of operations within a workcell.

SAGE, Sequence Analysis by Graphical Evaluation, is a logic programming framework for the analysis of repetitive sequencing problems within a robotics workcell[8,9]. SAGE takes as input the Constraints Net of an application problem and then generates a time-optimal sequence for all of the specified workcell operations (where this is feasible). After a preliminary analysis of the net structure and initial state defined by the user, the CN is used to explore, via heuristic search, the problem *Task space* which makes explicit all possible concurrency among workcell operations. Cycles within this space represent repetitive feasible sequences, from which the time-optimal one is selected. Finally, an executable program is constructed for the workcell runtime environment.

Although graphical aids have long been used for *simulating* robotic operations[5,11,18], only recently have *programming* systems emerged which incorporate a graphics interface. In the next section, we describe a few examples from the robotics literature.

2.1 Graphics and Robotics – related work

Until recently, most computing environments used for research made a clear distinction between *computing* and *graphics*, due to the dedicated nature of the available equipment; computer systems and graphics systems existed as separate devices. As a result, a recent survey[6] revealed that only a limited number of general systems for the analysis and simulation of Petri Nets support a graphical user interface. However, the growing popularity of integrated graphics workstations on the one hand, and the increasing sophistication of personal computers on the other hand, are helping to foster the integration of programming and graphics. For example, a graphical aid has been developed for programming a robotic arc-welding system[4]. Sketching with a mouse on a graphics display is used to 'program' the location of the welding seam, the position and orientation of the welding torch, and the cross-sectional geometry of the weld joint. An executable program is then generated automatically and down-loaded to the welding system for execution.

A sophisticated graphics system is described[13] for the 'programming' of a single robot interacting with its surroundings. The user creates a simulation program consisting of statements associated with the robot, its environment, and the graphical rendering. (Eventually, the robot commands are separated from the other statements and passed down to the robot controller for execution.) The editing and simulation are entirely menu-driven.

But closer to our work is the graphics interface for a Petri Net-like representation called the C-net used to model sequencing problems[12]. The C-net introduces functions associated with 'process input/output' and 'process status' to more conveniently model applications in process control and factory automation. The C-net is drawn directly on a graphics monitor using a light pen, by selecting nodes from a menu of graphic elements and then linking them on a grid. Clearly, the 'programming' effort here is small; the application problems involves simple sequencing, and therefore the user description takes a simple form.

But when the sequencing application also exhibits internal non-determinism, careful interaction must take place between the graphics interface and the user to 'extract' more knowledge about the problem at hand. In the next sections, we shall describe a simple graph structure called the *Constraints Net* which is automatically generated by an interactive graphical interface.

3. The Constraints Net

The Constraints Net (CN) is a graphical representation of of an application problem, based on the theory of Timed Petri Nets[15]. Operations (events) are represented by vertical bars called *transitions*, and the conditions which link the operations are represented by circles called *places*. These conditions are constructed from the state variables and their possible values as defined by the user. In addition, each transition has an associated fixed duration which corresponds to the execution time of the operation.

For example, consider a workcell configured for the repair of printed circuit boards, with a common jig shared by two robots to perform the repair. We might chose to define a state variable called "board_in_jig". Conditions can then be created for specific values of the state variables. For this example, if a board in the jig can be either "new" or "repaired", we would obtain [board_in_jig = new] and [board_in_jig = repaired].

Once the state variables and conditions have been defined, the operations can be specified. Each operation has four associated arguments: (i) a list of conditions called *pre-conditions* which enable the operation, (ii) a list of generic commands to be sequentially performed called *activities*, (iii), a list of conditions called *state changes* which are the consequences of the activities, and (iv) the durations of the activities.

When an operation consists of just one activity, we call it a *simple* operation; otherwise, we call it *complex*. Complex operations are useful when several related steps are enabled together and then follow one after the other. For example, a typical robot 'grasping' task involves at least these steps: approach, grasp, depart. However, it is convenient from the *user's perspective* to think of these steps as enabled by the same pre-conditions; thereafter, the steps are executed sequentially. Of course, distinct state changes can be associated with each activity. For example, if the object to be grasped was located in a part of the workcell shared by other robots, then the 'approach object' activity might cause the state change [shared_workarea = occupied], while the the 'depart' activity might cause the state change [jig = free].

This kind of modelling has proved to be adequate for our workcell sequencing problems since, as in most real-time applications, the execution times of events such as robot motions can be predicted quite accurately (apart from failure). Note that we are restricting our attention to deterministic operations.

In the next sections, we will describe how a graphical user interface can be used to interactively create and modify the CN in an easy way.

4. Graphics Details

4.1 Creating a new CN

To obtain an executable program for the runtime environment of the robotics workcell, the user must first define the application as a list of operations to be performed with their inter-dependencies made explicit.

The Graphics Interface (GI) is not menu-driven in the traditional sense, since a fixed menu of commands is always displayed (Table 1). User interaction specific to a particular item in the menu begins when that item is selected by the click of the mouse. Highlighting (blinking on/off) is used to visually confirm the selection of a given menu item.

For example, the selection of 'create_net' prompts the user for information required to initialize the graphics display for the creation of a new CN, such as the number of active elements in the workcell to be used. At this point, operations can be defined via the command 'add_node'. Again, command-specific interaction prompts the user to identify the active element, and then provide the pre-conditions, activities, state_changes, and duration associated with the corresponding workcell operations. Arcs are automatically generated from the conditions which comprise the pre-conditions and state_changes. The user is free to add nodes (to define the operations for each workcell element) in an arbitrary order, since the optimal sequencing of these operations will be automatically determined by SAGE.

A limited kind of consistency checking takes place as nodes are added, to make the user aware of partially-defined inter-dependencies among the operations, as when a condition appears in the pre-conditions of one operation but as yet in none of the state_changes of the other operations defined thus far.

4.2 The Display

The display is composed of a sub-window for text and a larger sub-window below for graphics. This graphics sub-window has three parts: a menu, an information header, and the CN itself; see Figure 1.

The graphical representation of the CN departs from the traditional Petri Net notation as follows. Nodes (operations) are drawn as rectangles, with an interior label in the form of a list eg. [1,2] meaning task # 2 of workcell element # 1. The workcell element number is perhaps redundant, since all tasks associated with the same element are automatically placed in a single column. However, these numbers are useful when generating or studying a large application, since the header will scroll off the display.

The importance of the task number is perhaps less obvious. Since operations can be entered by the user in an arbitrary way, the task numbers have no visible meaning. However, the sequence analysis to be performed requires that each operation be uniquely identified, so GI sequentially assigns all task numbers as the operations are entered.

Conditions which link operations can be thought of as arcs, with state_changes as sources and pre-conditions as sinks. But to avoid unnecessary visual complexity, arcs are displayed as triangular 'stub' icons entering or leaving the nodes.

When a condition is first added to the CN as a pre-condition or state_change, the arc stub is displayed as a diamond icon until the other end of the arc is defined; at that time, both stubs becomes triangles. Similarly, when a condition associated with a pre-condition or state_change is deleted, the arc stub at the other end is re-drawn as a diamond. In this way, the application programmer can always see at a glance which, if any, conditions are incompletely defined. A sample robot operation consisting of two activities $(Step_1, Step_2)$ is shown in Figure 2 along with its representation as a node. The conjunction of the conditions $Cond_1$ and $Cond_2$ enable both activities. The execution of $Step_1$ (duration 5 time units) makes the condition $Cond_3$ true, while $Step_2$ (duration 7 time units) makes both $Cond_4$ and $Cond_5$ true. The condition $Cond_1$ is shown as a diamond since it has not yet been defined as the state change of some other operation; all the other conditions are already completely defined (in terms of other operations not shown in the figure), and are therefore shown as triangles. Note that the conditions { Cond_i } all appear as separate arc stubs, although the logical groupings into pre-conditions and sets of state_changes are internally maintained by GI.

In Figure 3, we present a CN adapted from from research in our laboratory about the inspection and repair of hybrid integrated circuits and printed circuit boards[7]. There are three active elements: robot R1, robot R2, and a conveyor belt fixtured with two jigs.

Robot R1 is responsible for moving 'new' boards from the input_tray to a repair_jig, and then 'repaired' boards from the repair_jig to a checking_jig mounted on the conveyor. Robot R2 carries out the (pre-defined) repair using a dedicated repair tool, and the checking of the repair using a different sensor-based tool. The conveyor is responsible for moving the 'checked' boards out of the workcell; the unloading at the output_tray automatically puts the second jig, mounted on the other half of the conveyor, in the proper position for the checking operation. We assume that there is always a new board available in the input_tray, and that there is always room for another board in the output_tray.

Rather than associating synchronization with the boards which are cycling through the workcell, we define three multi-valued state variables in terms of the resident jigs (passive elements) as follows: board_in_repair_jig = { null, new, repaired }; board_in_checking_jig = { null, repaired, checked }; board_in_unloading_jig = { null, checked }. The 'null' value simply indicates that the corresponding jig is 'free' and in its default position.

The CN corresponding to the example as displayed by GI is shown in Figure 3. The corresponding CN in traditional Petri Net notation is shown in Figure 4.

In addition to selecting items from the command menu, the mouse can also be used to display information associated with the nodes and arcs. With a click of a button over the appropriate part of the display (edge of a node or arc stub), the relevant information is printed in the text sub-window. In the case of an arc stub, the stub selected by the mouse *and all the stubs associated with the same condition* are highlighted (displayed as filled polygons). This makes it easy to see the connections between related operations. The highlighting is terminated by clicking a different mouse button.

Once the CN is drawn, the programmer completes the description of the given workcell application by specifying the initial state of the workcell. Here, error checking is used to ensure that the initial conditions satisfy the pre-conditions of at least one operation. More complete error/consistency checking is described in the next section.

5. From CN to Workcell Program

Recall that multiple activities can be associated with a single operation (and therefore a single set of pre-conditions). This means that a multi-step operation can be described as one unit, which greatly simplifies the 'programming' effort. However, this complicates the analysis of the inter-dependencies among operations. Thus, once the CN is created by the user, the GI decomposes each 'complex' operation with multiple activities into a set of 'simple' ones with just one activity each. Extra conditions are automatically synthesized and added in pairs to the state_changes of each simple operation and the pre-conditions of its neighbour in the original sequence of activities, to ensure that these simple operations will be sequentially executed. Using the example in Figure 2, two separate operations would be created for the two activities. The pre-conditions as shown would be assigned to the new operation for $Step_1$, and a new condition would be synthesized to add to the state_changes shown for $Step_1$. This same new condition would then become the single pre-condition of the new operation for $Step_2$.

After this decomposition, SAGE performs a two stage analysis. First, certain graph-theoretic properties are investigated to ensure that the CN is consistent (each pre-condition re-appears in the list of state changes associated with some other operation, and vice versa) and deadlock-free. Second, the CN is used to explore, via heuristic search, the problem *Task space* which makes explicit all possible concurrency among workcell operations. Cycles within this Task Space represent repetitive feasible sequences, from which the time-optimal one is selected.

After this analysis, an executable program for the workcell runtime environment is constructed as follows. First, the pre-conditions and state_changes of the operations in the time-optimal sequence are combined with information about their corresponding activities to obtain a set of program 'skeletons', one per active workcell element. Then simple operations created by the decomposition of the complex operations in the original user description are re-assembled within each skeleton.

6. Implementation

GI is composed of a collection of programs written in G-Prolog[2], a superset of C-Prolog[14] which provides an interface to the SunCore graphics package[17] running on Sun-3 workstations under UNIX 4.3BSD. (SunCore is a particular implementation of the CORE graphics standard[1] developed by the

ACM.) One program creates and manipulates the display, another associates the display with its textual representation as a net, a third program performs the initialization of the mouse interaction, and a fourth serves as a common command interpreter. A fifth program performs the decomposition of operations with multiple activities, and a sixth program re-formats the CN for subsequent analysis by SAGE.

7. Conclusions

The complexity of modern robotic workcells, consisting of many cooperating elements (eg. robots, sensing systems), makes programming applications enormously difficult. In this paper, we have described a graphics interface (GI) to help cope with this complexity by organizing information in two levels. Information about the conditions associated with the operations (and therefore their interdependencies) is nominally hidden, to make evident the distribution of tasks among the active elements. The 'hidden' information associated with a condition However, a single click of the mouse over a condition highlights both where the condition appears as a precondition and where it appears as as a state change. GI also serves to guide the definition of the user application by (i) performing error checking and by (ii) displaying partially and completely 'defined' conditions using two different icons. In addition, the Constraints Net constructed from the user description is used to analyze the given application and obtain the time-optimal sequence of workcell operations. We are now extending our work to deal with applications which involve real-time sensing and data-driven tasks.

The authors gratefully acknowledge the financial support of FCAR and NSERC.

References

[1] "Status Report of the Graphics Standards Planning Committee, *Computer Graphics*, Vol. 13, no. 3, August 1979.

[2] B. Brachman, "GProlog User's Manual", Technical Report, Department of Computer Science, University of British Columbia, October 29, 1985.

[3] G. Carayannis, A. Malowany, "Improving the Programmability of Robotic Workcells", Proc. CG International '88 Conf., 1988.

[4] J.DeCurtins, J. Kremers, "SKETCH: A Simple-to-use Programming System for Visually Guided Robotic Arc Welding", Proc. IEEE Int. Conf. on Robotics and Automation", 1987.

[5] R. Dillman, M. Huck, "A Software System for the Simulation of Robot Based Manufacturing Systems", *Robotics*, Vol. 2, no. 1, March 1986.

[6] F. Feldbrugge , "Petri Net Tools", *Advances in Petri Nets 1985*, G. Rozenberg (ed.).

[7] P. Freedman, G. Carayannis, D. Gauthier, D., A. Malowany, "A Session Layer for a Distributed Robotics Environment", Proc. IEEE COMPINT '85 Conf., 1985.Springer-Verlag, 1986.

[8] P. Freedman, A. Malowany, " Sequencing Tasks within a Robotics Workcell: from Feasibility to Optimality", IEEE Pacific Rim Conf., Victoria, B.C., June 1987.

[9] P. Freedman, A. Malowany, " The Analysis and Optimization of Repetition within a Robot Workcell Sequencing Problems", Proc. IEEE Int. Conf. Robotics and Automation, 1988.

[10] M. Garey, D. Johnson, *Computers and Intractability: a guide to theory of NP- completeness*, Freeman and Co., 1979.

[11] F. Kahloun, A. Malowany, "A Robotics Workcell Simulator", Proc. IEEE COMPINT Conf., 1987.

[12] T. Murata, N. Komoda, K. Matsumoto, K. Haruna, "Petri Net-Based Controller for Flexible and Maintainable Sequence Control and its Applications in Factory Automation", IEEE Trans. on Industrial Electronics, Vol. 30, no. 1, February 1986.

[13] A. Naylor, L. Shao, R. Volz, R. Jungclas, P. Bixel, K. Lloyd, "PROGRESS: A Graphical Robot Programming System", Proc. IEEE Int. Conf. on Robotics and Automation", 1987.

[14] F. Pereira, "C-Prolog User's Manual", Dept. Architecture, University of Edinburg, February 1984.

[15] C. Ramchandani, Project MAC Technical Report TR-120, MIT, 1974.

[16] W. Reisig, *Petri Nets – an introduction*, Springer-Verlag, 1985.

[17] SunCore Reference Manual, Part No: 800-1257-03, Revision G of 17, Sun Microsystems, February 1986.

[18] H. Wörn, G. Stark, "Robot Applications Supported by CAD Simulation", *Robotics and Computer Integrated Manufacturing*, Vol. 3, no. 1, 1987.

Item	Description
create_net	begin a new CN
new_net	load a new CN from a file
scroll_up, down	scroll display
save_net	save current CN and prompt for initial state
add_node	add new node and info to current CN
delete_node	delete a node from the current CN
edit_node_info	change pre-conditions, etc.
exit_mouse	quit the mouse interaction
go_pick	restart the mouse interaction
help	print this table of commands
halt	exit from the graphical interface

Table 1: The menu of user commands available via the mouse.

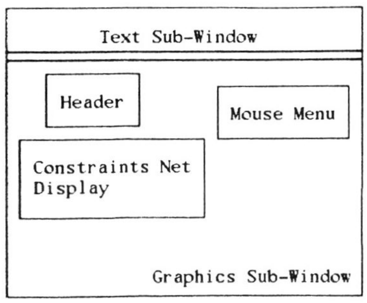
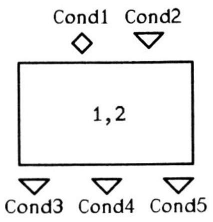
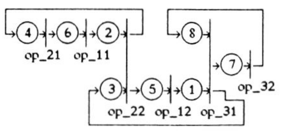


Figure 1: The display architecture of the Graphics Interface.



Item	Description
Pre-conditions	$\{Cond_1 \text{ AND } Cond_2\}$
Activities	$\{Step_1, Step_2\}$
State_changes	$\{(Cond_3), (Cond_4, Cond_5)\}$
Durations	$\{5, 7\}$

Figure 2: A typical operation displayed as a node



Place (condition)	Description
1	board_in_checking_jig = checked
2	board_in_repair_jig = repaired
3	board_in_checking_jig = null
4	board_in_repair_jig = null
5	board_in_checking_jig = repaired
6	board_in_repair_jig = new
7	board_in_unloading_jig = checked
8	board_in_unloading_jig = null

Transition (activity)	Description
op_{11}	repair
op_{12}	check_the_repair
op_{21}	move_board_from_input_tray_to_repair_jig
op_{22}	move_board_from_repair_jig_to_checking_jig
op_{31}	move_checking_jig_to_unloading_position
op_{32}	unload_board_into_output_tray

Figure 4: The internal representation of the CN corresponding to Figure 3. The Petri Net notation makes explicit the conditions and how they link related activities.

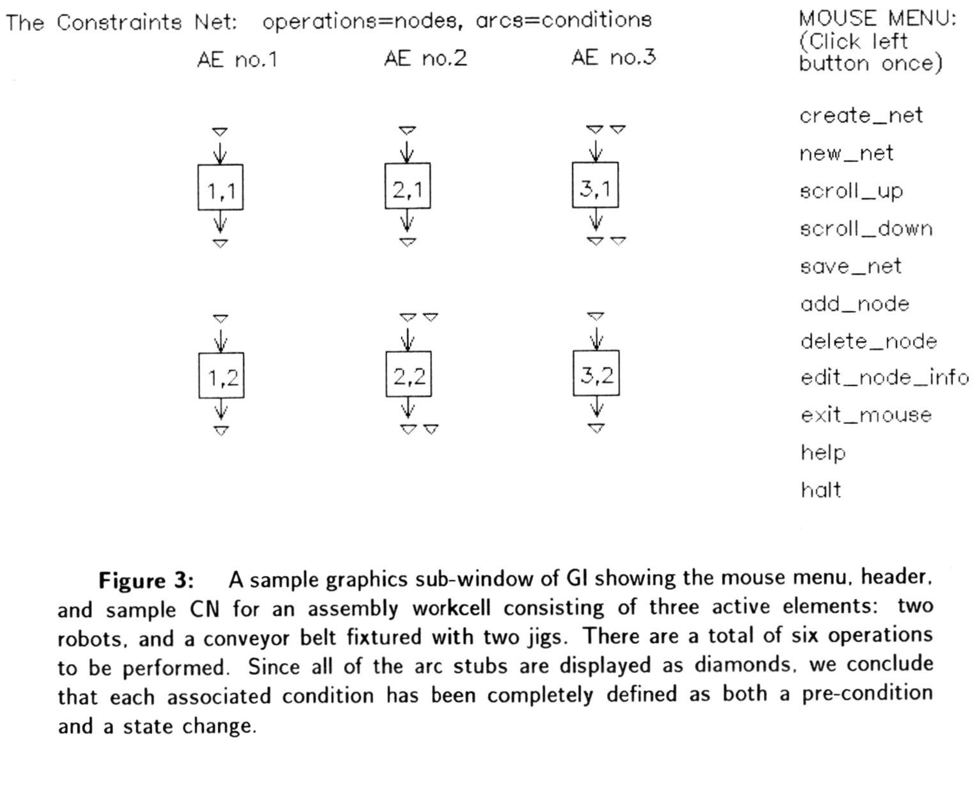


Figure 3: A sample graphics sub-window of GI showing the mouse menu, header, and sample CN for an assembly workcell consisting of three active elements: two robots, and a conveyor belt fixtured with two jigs. There are a total of six operations to be performed. Since all of the arc stubs are displayed as diamonds, we conclude that each associated condition has been completely defined as both a pre-condition and a state change.