ROBOT GRAPHIC SIMULATION TESTBED
Final Report
National Aeronautics and Space Administration
Marshall Space Flight Center

Prepared by:
Dr George E. Cook
Dr Janos Sztipanovits
Dr Csaba Biegl
Dr Gabor Karsai
James F. Springfield

Department of Electrical Engineering
Vanderbilt University
Nashville, TN 37235, USA

Contracting Officer Representative: Dr Kenneth R. Fernandez

NASA Grant No. NAG8-690

August, 1991

Contents

1 Executive Summary 4

2 Introduction 6

3 Workstation Implementation of ROBOSIM 7
- 3.1 The HP350SRX Graphics Workstation . . . 7
- 3.2 The MD Program . . . 10
- 3.3 The R2 Program . . . 12
- 3.4 Simulation Library and Environment . . . 28
- 3.5 Inverse Kinematics . . . 29
- 3.6 Collision Detection . . . 30
- 3.7 Example: Surgical Positioner . . . 35
- 3.8 Port to the Intergraph Workstation . . . 39

4 Intelligent Graphic Modeling Environment 41
- 4.1 Critique of the Basic Graphical Modeling Technique . . . 41
- 4.2 System Design of the Graphic Simulation Environment . . . 42
- 4.3 *Agent* command interface . . . 44
  - 4.3.1 Command format . . . 45
  - 4.3.2 Error reporting . . . 45
  - 4.3.3 Object creation commands . . . 47
  - 4.3.4 Object transformation commands . . . 47
  - 4.3.5 Composite objects . . . 47
  - 4.3.6 ROBOSIM objects . . . 48
  - 4.3.7 Agents . . . 48
  - 4.3.8 Object removal . . . 48
  - 4.3.9 Agent positioning . . . 48
  - 4.3.10 Position reporting . . . 51
  - 4.3.11 Grasping . . . 51
  - 4.3.12 General graphics setup . . . 52
  - 4.3.13 General commands . . . 52
- 4.4 Automation Interface for Robot Modeling Systems . . . 53
- 4.5 The HDL System . . . 64

4.5.1 Introduction . . . 64
4.5.2 Semantics of HDL . . . 65
4.5.3 Declaration of Primitive Modules . . . 65
4.5.4 Declaration of Compound Modules . . . 67
4.5.5 HDL Programmatic Interface . . . 70
4.5.6 Monitor . . . 72
4.5.7 An example . . . 74
4.6 Interfacing of HDL to *Agent* . . . 77

5 Case Studies **78**
5.1 Space Station Modeling Using ROBOSIM . . . 78
5.2 Operational Modeling of the Space Station . . . 80
5.3 Study of the Space Station ECLSS . . . 87
5.3.1 Objectives of the ECLSS study . . . 88
5.3.2 Model-based diagnostic system . . . 89
5.4 ECLSS Study: Diagnostics and Repair . . . 93
5.4.1 Process and Fault Modeling for the ECLSS . . . 93
5.4.2 Hierarchical Process Model (HPM) of the ECLSS . . . 93
5.4.3 Declarative Form of the HPM . . . 95
5.4.4 Hierarchical Fault Models (HFM) for the ECLSS/PWP . . . 95
5.4.5 Definition of repair actions . . . 97
5.4.6 Integrated monitoring and diagnostics with robot simulation . . . 99

6 Suggestions for Future Work **101**

A Structure Declarations for the Simulation Library **103**

B Simulation Library Functions **107**

C HDL/C Interface **110**
C.1 HDL/C Interface . . . 110
C.1.1 HDL Parameters . . . 110
C.1.2 HDL Context Tables . . . 111
C.1.3 Preparing and loading "Plain" C scripts . . . 112
C.1.4 Preparing "Embedded" C scripts . . . 115

List of Figures

3.1 Creating a custom object 20
3.2 Cylinder and custom object before and after attachment 21
3.3 Base link with joints being checked 22
3.4 Base link being saved and compiled by ROBOSIM 23
3.5 First link being saved and compiled by ROBOSIM 24

4.1 Main functional components of the simulation environment 43
4.2 Layers of the Multigraph Architecture 56
4.3 Structure of the Multigraph Architecture 57
4.4 Graphical model for a reconfigurable controller 60
4.5 Structure of the MEE 63

5.1 Solar panels - I. 80
5.2 Solar panels - II. 81
5.3 Servicing Robot 82
5.4 Middle truss assembly 83
5.5 Space Station Model 84
5.6 Solar Panel Motion 85
5.7 A Hierarchical Process Model 91
5.8 Fault Propagation Digraph of a Process 92
5.9 Process Hierarchy of the ECLSS 94
5.10 Declarative form of the AirControl Process 96
5.11 Fault Diagnosis Declarative Form 98
5.12 Integrated 3D models and fault monitoring/repair system 100

Chapter 1

Executive Summary

The objective of this research program was twofold. First, the basic capabilities of ROBOSIM (a graphical simulation system developed jointly by NASA-MSFC and Vanderbilt University) were improved and extended by taking advantage of advanced graphic workstation technology and artificial intelligence (AI) programming techniques. Second, the scope of the graphic simulation testbed was extended to include general problems of Space Station automation.

The first objective is a logical continuation of the joint NASA/Vanderbilt ROBOSIM development. State-of-the-art graphic workstations offer new opportunities for simulation of complex, linked geometrical structures. Hardware support for 3-D graphics and high processing performance make high resolution solid modeling, collision detection, and simulation of structural dynamics computationally feasible. With the introduction of new AI programming techniques, graphic structural simulation can be combined with high-level, AI-based control functions; thus the simulation testbed can support studies in task level planning and in other issues of autonomous control.

The Space Station is a vastly complex system with many interacting subsystems. Automation, being a decisive factor in crew productivity and safety, is expected to play a major role in the Space Station operation. The rationale for the second objective of this project is based on the fact that formulation and testing of automation concepts require understanding the behavior of and the interactions among the various subsystems. For example, the Environmental Control and Life Support System (ECLSS), which is one of the most complex subsystems in the Space Station, is an aggregate of interdependent mechanical, chemical and electrical processes. These processes interact with each other and impose constraints on the operation of other subsystems in many levels. The following list includes a few examples for these interactions:

- the air temperature control in the ECLSS is directly related to the Thermal subsystem,
- the ECLSS is one of the major electric energy consumers in the Space Station,
- therefore its operation interacts with the Electric Power Supply subsystem,

- effects of the ECLSS operation on the utility consumers (air, potable water, hygienic water, wash water, etc.),
- waste material removal may interact with low-gravity experiments.

Design and testing of automation concepts demand modeling of the affected processes, their interactions and that of the proposed control systems. These models may vary in objective and sophistication, in accordance with the level of control functions to be studied. The analysis of elementary control loops that maintain the value of a process variable require the use of high fidelity dynamic simulation. The testing and validation of higher level and autonomous controllers will necessitate the use of AI-based models representing qualitative as well as quantitative features of processes. Extended modeling techniques include the explicit description of hierarchical process structures, causal relations, fault propagation models and component hierarchies.

The automation testbed was designed to facilitate studies in Space Station automation concepts. Its main purpose is to provide cost-effective solutions for the analysis of the interactions among work packages, and for experiments with the scars and hooks provided by the IOC automation concepts for advanced automation. Supplementing the ROBOSIM graphical simulation package with the required new capabilities is a complex task. It requires significant extension of the system in many ways, including the incorporation of AI-based modeling tools, the application of automatic program generation facilities for fast prototyping, and the introduction of advanced software engineering techniques for managing large-scale models.

In this Report, the steps of this process are discussed. In the first section the new capabilities of the graphic workstation version of ROBOSIM are described. The work accomplished in the first year of the project has resulted in significantly improved 3-D graphic capabilities, interactive model building tools, and a solution for collision detection. The second section discusses the design details and implementation of a new graphical simulation package. The new design makes it possible to integrate the system with tools supporting automation studies as well. The third section provides case studies that demonstrate the usage and capabilities of an integrated structural modeling and automation testbed. The case studies include the structural model of the Space Station, and a process and failure model of the ECLSS Potable Water Processing subsystem coupled with the geometrical modeling environment. The fourth section describes some suggestions for future research and development work.

The work described in this paper has been mostly performed on a Hewlett Packard 9000/350 SRX graphics workstation and on an Intergraph graphics workstation. We would like to express our gratitude to the Industrual Application Center of the Hewlett Packard Company and Intergraph Corp. for their support which made this research possible.

Chapter 2

Introduction

This report is organized into five chapters. Chapter 3 which follows the introduction describes the work done in porting ROBOSIM to the HP350SRX graphics workstation. New additional ROBOSIM features, like collision detection and new kinematics simulation methods are also discussed here. The chapter concludes with a brief description of the ROBOSIM port to the Intergraph workstation.

Chapter 4 can be divided into three parts. In the first part - based on the experiences of the work on ROBOSIM - we describe a new graphics structural modeling environment, which is a part of a new knowledge-based multiple aspect modeling testbed. The second part of the chapter contains a description of the knowledge-based modeling methodologies and tools already available to us. The chapter concludes with the description of a model-based package which can be used for designing and simulating robot controllers, together with the graphics modeling environment mentioned above.

Chapter 5 contains three case studies in the area of Space Station automation. First a geometrical structural model of the station is presented. This model was developed using the ROBOSIM package. Next the possible application areas of an integrated modeling environment in the testing of different Space Station operations are discussed. One of these possible application areas is the modeling of the Environmental Control and Life Support System (ECLSS), which is one of the most complex subsystems of the station. Using the multiple aspect modeling methodology presented in Chapter 3 we built a fault propagation model of this system and integrated it with the geometrical modeling environment: this is described at the end of the chapter.

Chapter 6 concludes the report by suggesting possible future research directions for the application of these modeling techniques in automation systems.

Chapter 3

Workstation Implementation of ROBOSIM

This chapter describes the work which has been done in order to enhance the capabilities of the ROBOSIM graphical structure modeling package. ROBOSIM in its original form was a command-oriented modeling language, with a not too user friendly programming interface. Furthermore its graphics capabilities were limited, due to the fact that originally it was designed for use on a remote graphics terminal attached to a VAX-like processor, which did not offer many of the features available on modern graphics engineering workstations. Further additions include simulation libraries for collision detection and dynamics, which are also described later in this chapter.

3.1 The HP350SRX Graphics Workstation

Since part of the impetus for extending ROBOSIM was the capabilities provided by graphics workstations, it is necessary to understand these capabilities. All of the information that follows is specifically oriented towards the HP350SRX workstation; however, much is generally applicable to other workstations.

The HP350SRX workstation has a pixel resolution of 1280x1024 pixels. There are 16 image planes and 4 overlay planes. Each plane is one bit per pixel. Typically the image planes are used for graphics and the overlay planes are used for XWindows. When the image planes are rapidly changed (animation) flickering results if the images are not double buffered. This means that only 8 image planes are actually available. While one set of image planes are being displayed, the other set is being changed. Then, the sets are switched. This results in flicker-free motion at the cost of reduced numbers of colors. Since only eight planes are used at one time, only 256 colors can be displayed at one time. The overlay planes, if used, will hide the image planes. Therefore, if X is being used, a transparent window is created. This allows X applications to be run while seeing what is in the image planes.

The most important capability of the workstation is the increased speed and facilities

provided by the Starbase graphics library and the hardware graphics accelerator. These facilities allow display of three dimensional graphics objects with options such as hidden surface removal, shading, perspective views, and colors.

The hardware accelerator includes a matrix multiplier. This allows multiplication of 4x4 matrices much faster than could be done in software. This facility is used to a great extent in display of objects. There are many coordinate transformations occurring during display such as rotation and translation of objects in modeling coordinates, conversion of modeling to world coordinates, perspective transformations, world to virtual device coordinates, and virtual device to device coordinates. Each of these involves multiplying by matrices; also, there can be many levels of transformations in modeling coordinates. All graphic objects are "put through the pipeline" of transformations, and the hardware multiplier is a key part in providing real-time speed. There is one difference between the transformation matrices used in Starbase and the ones traditionally used by roboticists. The graphics standard uses matrices that are the transpose of the ones used in robotics. Therefore, all matrices in the programs are represented in the graphics standard form. For this reason, all matrix equations had to be the reverse of those used in robotics.

Another useful feature of Starbase is the display list. A display list is made up of segments. Each segment can be thought of as a procedure. One segment can call another and the called segment returns to the calling segment when finished. Almost any Starbase function can be placed in a segment. Then, whenever that segment is traversed those commands are executed. This is very useful; for instance, all of the Starbase polygon procedure calls that make up a robot link can be placed in a segment with a transformation matrix that represents the transformation resulting from a particular value for that link's joint variable. Then, changing the transformation matrix in the display list will result in that link "moving" the next time that display list is traversed. A segment network is shown below. It has been printed from the simulation program for an actual robot. It has been abbreviated in parts. "fd" is the file descriptor returned by Starbase when a display is opened for graphic output. The {} indicate an array that has not been printed out. Segment #0 is the main segment. It has a call to segment #1. Segment #1 is for a robot. If there were another robot, then there would be another call in segment #0. Segment #1 first pushes a matrix onto the transformation stack. This transformation corresponds to the position of the robot in the world. Next, segments 2 – 9 are called. In segment #2 the first `concat_transformation3d` is a transformation describing the structure of the link. The second transformation describes the current value of the joint variable. Concat multiplies the matrix by whatever is currently on the transformation stack and pushes the result back on the stack. Now, the polygons in segment #2 are displayed after first being transformed by whatever is on top of the transformation stack. Segment #2 then returns control to segment #1, and traversal continues through segment #9. When segment #9 returns, the top of the matrix is popped off and returned to the state it was in before traversal began.

```
segment 0 begin
    move3d( fd, 0, 0, 0)
```

```
    dl_label( fd, 1)
    call_segment( fd, 1)
segment 0 end

segment 1 begin
    push_matrix3d( fd, {})
    call_segment( fd, 2)
    call_segment( fd, 3)
    call_segment( fd, 4)
    call_segment( fd, 5)
    call_segment( fd, 6)
    call_segment( fd, 7)
    call_segment( fd, 8)
    call_segment( fd, 9)
    pop_matrix( fd)
segment 1 end

segment 2 begin
    concat_transformation3d( fd, {}, 0, 0)
    concat_transformation3d( fd, {}, 0, 0)
    polygon3d( fd, {}, 5, 1)
.     .     .
.     .     .
    polygon3d( fd, {}, 5, 1)
segment 2 end

    .  .    .    . .
    .  .    .    . .

segment 9 begin
    concat_transformation3d( fd, {}, 0, 0)
    concat_transformation3d( fd, {}, 0, 0)
    polygon3d( fd, {}, 5, 1)
       .    .  .
       .    .  .
    polygon3d( fd, {}, 5, 1)
segment 9 end
```

The ability to pick an object that is displayed on the screen is a very important part in the graphics editor. Starbase provides a simple way to do this. When a display list is displayed on the screen the points making up an object are eventually converted to actual device coordinates. Now, given a range of coordinates, Starbase can return information regarding what is displayed in that range. This consists of the segment number, the most

recent label within that segment (if any), and the offset from that label. For instance, if the first polygon in segment #9 fell within that window, then Starbase would return segment #9, label #0 (there is no label in segment #9), and an offset of 3 (the first polygon is the third command in segment #9). Starbase can even return the entire path through the display list, giving all called segments and offsets leading up to the polygon in segment #9.

XWindows provides the ability to read the mouse position. A program can read the position of the mouse and convert that position to the form required by Starbase. Thus, one can use the mouse to point to an object on the screen, and a program can figure out what is being pointed to.

XWindows also provides many other facilities that proved to be useful in implementing this work. One of the most useful aspects of X is the menus. Using X, one can implement menus very easily. This allows user-friendly interfaces to be written without having to deal with the complexities introduced. For instance, a set of menus can be created to manipulate some display list. The menu entries are created and X is told which procedures to execute upon selection of the corresponding menu entry. A transparent window in the center of the screen allows the image planes (graphics planes) to be seen through the X application.

There are also two other peripherals which have been extensively used. The button box and the knob box provide very easily used input capabilities. Once the devices have been opened for use by a program it is quite simple to poll them. The button box returns an integer corresponding to the button pushed, if any. The knob box is just as simple to poll, but it has additional features. The knob box has nine knobs and each can be set differently. A knob's range can be set; for example, a knob can return a number between -1. and 1. or it can return a number between 10. and 100. Also, the knob can be preset to a particular value. This means that whatever position the knob is in, that position corresponds to the set value.

The features of the HP350SRX workstation make it ideal for use in high-performance graphics applications. The resolution and color capability allow for sophisticated graphics. The graphics accelerator provides speed, and the display lists provide easy access to graphics hardware. XWindows allows friendly and generic user interfaces to be written simply and easily. And the peripherals such as the mouse, button box, and knob box provide a flexible and diverse range of input.

3.2 The MD Program

Porting ROBOSIM to the HP350SRX workstation added no additional features to those found in the VAX version. ROBOSIM on the HP no longer used the TEKTRONIX 4014 interface, although XWindows allowed certain windows to operate in a TEKTRONIX emulator mode. ROBOSIM was adapted to use the Starbase graphics move and draw commands. ROBOSIM still performed all transformations internally, but used a window from X and Starbase graphics for output. This allowed one window in which to run the process and another in which to see the output. This capability spurred an early attempt

at allowing an interactive mode of operation in which ROBOSIM commands were typed in, and the effects were immediately seen in the display window. However, this method was never effectively implemented or used.

The MD program originally evolved as a means of displaying a robot that had been generated by ROBOSIM. Through this program, a user could display a robot and set colors and other attributes such as hidden surface removal, shading, and specular reflection. Also, the camera position (i.e. the position from which the object is looked at) could be changed to provide views of the object from many different perspectives.

Extensions to the basic MD allowed multiple robots and objects, and it even has provisions to accept joint angles and other parameters from a separate process. With this feature, a primitive simulation can be run. An early use of this involved a lisp process piping commands to a space station model that would orient the solar collectors to receive maximum exposure. MD was also able to run in a mode in which joint angles were read from a file and the robot's joints were cycled through these. This feature was used for simple simulations of downhand welding. Two robots, one a six degree of freedom robot, and the other a two degree of freedom positioner, were simulated. The robot performed the welding and the positioner assisted in maintaining the downhand position and proper orientation of the wire feed to the direction of movement of the torch. The joint angles were generated by a separate program and stored in two files. Using MD, one could look at the robots from various positions to visually verify that the downhand welding was working correctly.

The basic structure of MD involves loading the link files of a robot and creating display segments corresponding to each link. Each segment has a transformation matrix and a polygon list. Also, there is a segment for the entire robot that has a transformation matrix describing the position of the robot in the world, and commands that set the color and other parameters for the robot. The input devices for MD are the button box and the knob box. These devices provide the ability to turn functions of Starbase on and off and to adjust parameters of Starbase. For most functions, there is a one-to-one correspondence between Starbase functions and MD functions. MD is useful for looking at a robot after it has been made by ROBOSIM. The robot can be brought up on the screen, looked at from various positions, and the joints can be moved.

The capabilities of MD for more complicated simulation were very limited, and further work on MD was replaced by the development of the simulation library and environment. MD is still used for photographing robots and other structures such as the space station. It is also still used for quickly verifying robots or other structures that have been constructed with ROBOSIM. Although it is not used directly for simulation purposes now, the components of it dealing with graphics manipulations are still used in R2 and other programs.

3.3 The R2 Program

The development of R2 arose from the capabilities provided by MD and the need for an easier to use and more flexible interface to ROBOSIM. R2 was designed to overcome some of the limitations of ROBOSIM while taking advantage of the facilities available on graphics workstations. However, complete compatibility with ROBOSIM was desired; this was accomplished by the output of R2 being ROBOSIM code. Having R2 generate ROBOSIM code allowed R2 to be much simpler. It was not necessary to reimplement what ROBOSIM already provided. This method has proved to be the most flexible. Now, robots can be designed by writing a ROBOSIM program, using R2 to generate a ROBOSIM program, directly generating files from custom programs, or any combination thereof.

ROBOSIM provides a simple way in which to design robots. Based on the specifications in a user-written 'program' a file for each link is generated. This file contains the vector list that is used to draw the robot, the A-matrix, the Denavit-Hartenburg parameters, joint types, and the pseudo-inertia matrix. However, this method requires the user to maintain a lot of information that the computer can handle much more easily. Since ROBOSIM creates every object at the origin, the user must keep track of each objects' dimensions in order to place it such that it will be in the proper position and orientation with respect to the other objects in a link. The only other method that ROBOSIM allows is to load in data files that have been generated by some other method. This requires a custom written FORTRAN program with appropriate calls to ROBOSIM functions. This is the most flexible way in which to use ROBOSIM, but also the most difficult. What is needed is a flexible, but user-friendly, environment in which to design robots.

Before discussing the internals of R2, it is useful to see how it works from a user's point of view. What follows is basically a user's manual for R2. However, some knowledge of ROBOSIM is expected. For information see the ROBOSIM manual and tutorial. It is recommended to read the following while running R2. Proper execution of all capabilities requires the proper setup of several files and directories. This is explained in the ROBOSIM manual. Execute R2 from your 'source' directory.

First, R2 is designed to run under XWindows. Therefore, type xstart to run XWindows. To execute the program type: `r2 [-t terminal] [-m message_level]`. The default terminal type is "hp98721". The only other terminal currently recognized is a "hp300h". The `message_level` refers to the amount of help that is available; the default level is level 0. At this level only error messages are displayed. At level 1, a small window is created in the upper left corner. then, whenever the program is waiting for input from the user, an appropriate message is displayed. Level 2 is the highest level; after any menu item is selected, a window with information describing the command is displayed. When the information has been read, the user clicks the mouse on the "OK" button. The user interface consists of the graphics window, where the model is displayed, a line of menus across the top, a diagram showing the current meaning of the buttons, and a diagram of the knobs showing their meaning. The use of the button box and knob box in R2 is the same as that in MD.

- **Mouse:** R2 is designed to make extensive use of the mouse. The only time at which the user uses the keyboard is when it would be more difficult to use the mouse. This only occurs when requesting a file name for the robot, or environment. At all other times, input is received from the mouse, the button box, or the knob box. To select a menu move the mouse's cursor until the desired menu heading is highlighted. Now, press either of the mouse's buttons and hold it. The menu will appear below. While holding the button down, move the cursor down the menu until the desired menu entry is highlighted; then release the button. If (before releasing the button) you decide that the wrong menu has been selected, move the cursor out of the menu and release the button. If you have already selected a menu item, most functions provide a means to cancel them with no effects.

- **Numeric input window:** Many functions make use of this window. It consists of the numbers 0-9, a decimal point, a minus sign, CANCEL, END, and a set of parameters (such as RADIUS and HEIGHT for a cylinder, or X,Y, and Z for translate). When invoked, all parameters are initialized to zero. However, all objects that are made must have positive values. To select a parameter move the mouse cursor over the desired parameter and press the left button on the mouse. Then use the mouse to enter the desired value. If you make a mistake, simply press the parameter "button" again, which will set the parameter to zero and allow you to reenter that parameter. When finished entering parameters, select END. If all is well, the command will be executed. If at any point you decide to abort this command, then press the CANCEL button in the window.

- **Quit menu:**

  Quit: Exits from the program.

  Restart: Deletes the current model from memory, but does not exit the program.

- **Make object menu:**

  Box: Uses the numeric input window (described above). This command has three parameters: x, y, and z. These three parameters are the dimensions of the box along the three coordinate axes.

  Cylinder: Uses the numeric input window (described above). This command has two parameters: RADIUS and HEIGHT. The cylinder is created with height along the z-axis.

  Cone: Uses the numeric input window (described above). This command has two parameters: RADIUS and HEIGHT.

  Truncated cone: Uses the numeric input window (described above). This command has three parameters: UPPER RADIUS, LOWER RADIUS, and HEIGHT.

Sphere: Uses the numeric input window (described above). This command has one parameter, the RADIUS of the sphere.

Special surface: h This command is used for creating custom objects. You do this by first creating a polygon and then extruding or revolving it to create a solid object. The right button selects the starting point. The left button draws a line. To adjust the scale push the scale button, and enter a value at least two times the amount of your largest coordinate. The resolution is useful for specifying the smallest unit that will be differentiated. If every point is a multiple of five, then set the resolution to five. (Special note: you must define the polygon in a counterclockwise direction for extrude and clockwise for revolve.) Warning: due to implementation constraints in the simulator's collision detection algorithm, all polygons must be convex. at this time no correction or detection of concave polygons is made, so it is the responsibility of the user to provide this check.

Clone: Allows the copying of an object. This is especially useful for copying the custom designed objects since they require the most work. After selecting clone, select the object to be cloned, with the mouse.

- **Manip object menu:**

Translate: Uses numeric input window (see description above). This command has x, y, z, and HOME for parameters. Translations are relative (i.e. they occur relative to the current position). To return an object to its home position, press HOME, "1", and END.

Rotate absolute: Uses numeric input window. However, the x,y, and z here represent rotations around the corresponding axes. Rotations are absolute, not additive. If you specify a rotation on an object, and then later another rotation, the first rotation is lost and the new rotation is from the objects' home position.

Rotate relative: Same as rotate above, except that these rotations are from the current position.

Delete: Waits for you to select an object for deletion. Use the mouse to select the object. Pressing the left button of the mouse while not on an object cancels this command.

Attach: Lets one object be attached to another object. First, use the mouse to select the base object, then select the polygon of the base object where the attachment is to be. Then, select the object to be attached and finally the polygon of the attached object. This command will attach the two objects selected such that the two polygons selected line up. This attachment creates a hierarchy such that the movement of the base object occurs to the attached object, but a movement of the attached object will not affect the base object. The new home position of the attached object is its position as attached to the

base object. Once an object is attached it can not be unattached. The object must be deleted and made again.

Resize: Lets an object be resized. It is especially useful along with the attach function. If several objects are created and attached together, then any of them can be resized and the relationship between them will be maintained. After selecting resize, the object to be resized is selected with the mouse. Then a window identical to the one used to create it appears. Enter the new dimensions, select END and the object will be resized.

- **Links menu:**

Revolute joint, **Prismatic joint**, **Fixed joint:** These three commands create a joint of the corresponding type. After selecting an entry the user is prompted to select whether it is to be an i-joint or an i+1-joint. An i-joint is the place of attachment to the previous link, and an i+1-joint is the place of attachment to the next link.

Rotate, **Rranslate**, **Relete**, **Attach:** These commands operate just like the ones in the "Manip object" menu. The reason to have separate commands for joints is that it is difficult to select them on the screen with the mouse.

Check joints for validity: This command checks the relationship between the i and the i+1 joint to make sure that it follows the Denavit-Hartenberg convention, as required by ROBOSIM.

- **File Management Menu:** This menu provides three basic capabilities: save a session, load a session, generate ROBOSIM code, and run MD.

Save file: This command saves the current model. The user is prompted as to whether it is to be saved as an environment file, a link file, or to exit this command. Then the user is prompted for a robot name and then for an extension.

Load file: This command loads a previously saved model. The user is prompted in the same way as save file above.

Generate ROBOSIM File: This command prompts first as to whether the file to be generated is for a robot or environment. Then the name is asked for. The ROBOSIM file is then generated, ROBOSIM is called and the file is executed. Control is then passed back to R2. The robot or environment can now be viewed by MD, if R2 is running on an hp98721 display.

MD: This command executes the MD program. This allows the robot to be viewed completely. Does not work with environment files presently. Also, can not be executed on an hp300h.

- **HP300 Menu:** This menu implements some functions on the hp300h. Since this machine does not have the button box or knob box it is necessary to implement them this way.

Look From: This command uses the numeric input window. Specify the X, Y, and Z coordinates to look from. At least one must be non-zero.

Look At: This command uses the numeric input window. Specify the X, Y, and Z coordinates to look at.

Although the interface gives the appearance of an object oriented structure, it is not implemented in this manner. The basic structure in this program is an array of pointers. Currently, this is set to a size limit of 100. This means that the most objects that can be in one link is 100 primitives. However, this number can be set to anything and the program recompiled. A better structure would be a linked list of objects that is dynamically allocated. At present, however, this method has not been a problem. The actual C structure declarations are shown below.

```
#define MAXOBS 100
#define MAXKIDS 10

typedef struct vertex {
    float x;
    float y;
    float z;
    float md;
} vertextype;

typedef struct {
    int s_p;      /*source polygon*/
    int d_o;      /*destination object*/
    int d_p;      /*destination polygon*/
} childtype;

typedef struct object {
    char  *name;
    int   type;
    int   vertices;
    vertextype *model;
    int   custom_vertices;
    float custom_extrude;
    vertextype *custom_model;
    float size[5];
    float amat[4][4];
    float ref[4][4];
    int   display_list;
    childtype kids[MAXKIDS];
} objecttype;
```

Whenever an object is created, enough memory for a structure of type objecttype is allocated and the pointer to this memory is stored in the array. All information relating to a particular object is stored in this structure. The first element in this structure is the name of the object. This name is actually the ROBOSIM command that is used to generate this object (i.e. BOX, R-JOINT-I). The name also directly corresponds to the next element: the type. The type is an integer that represents the ROBOSIM command. The variable 'vertices' is the number of points in the model. The variable 'model' is a pointer to the list of vertices that describe the graphic model. The `custom_vertices` is the number of points in the polygon that is used to generate a custom surface (REV-SURFACE and EXTRUDE-SURFACE). `Custom_extrude` is used in an extrude-surface object; it is the amount the object is extruded. The `custom_model` is a pointer to the polygon that is used in a custom surface. The 'size' array is an array of parameters that can be used as the arguments to primitive calls. For instance, if the object is a box, then the first three elements of 'size' will be used to store the x,y, and z dimensions. The 'amat' variable is a matrix representing a transformation (rotation and translation) on the object. The 'ref' variable is also a transformation, but it is used to define the home position of the object. The `display_list` variable is an integer that is the descriptor of the `display_list` in which this object is stored. The `display_list` is a set of graphics functions that when traversed will result in the graphic object being displayed. The 'kids' array is an array describing the children of an object. One object becomes another object's child when the child object is attached to the parent object. Currently, the maximum number of children one object can have is ten. However, this value can be changed. Each child is described by three integers. The first, `s_p` or source polygon is the number of the polygon of the parent where the child is attached. The `d_o` or destination object is the array index of the object that is attached. The `d_p` or destination polygon is the polygon of the child object that is attached to the parent.

After an object is selected from the menus and the parameters have been entered, the object is created. The FORTRAN code that generates the primitives in ROBOSIM is also used in R2. The use of the same code ensures that what is seen in the editor is the same as what will be by ROBOSIM. The FORTRAN routines store the vector lists in an array that is passed to them. After getting this information, the editor stores it in the structure allocated for the object and in a slightly different form in a display list. The other variables in the structure are filled out, the transformation matrices are set to identity, and a call to the newly created display list is inserted into the root display list. Now, the next time the `display_list` is traversed the object will be displayed.

Once an object has been created (i.e. an instance is made of the object), 'messages' can be sent to it. From the user's point of view, this is what is done. However, the implementation is different. The object is selected by picking it with the mouse. R2 waits for a mouse button to be pressed and then reads the (x,y) location of the mouse. These coordinates are then used by Starbase to determine what primitive is in that area. Starbase returns the display list number, a label number (if any), and the offset from the label. With this information, R2 can decide which object and polygon have been selected.

Translations and rotations result in changes to the transformation matrix: 'amat'. A matrix representing the appropriate translation or rotation is made and then multiplied by 'amat'. The result is put back into 'amat'. The new 'amat' also replaces the old matrix in the display list.

Attaching an object to another object is a complex procedure. First the object to be attached (child object) is selected and then the polygon attachment point is selected. The same is done for the base object. R2 then knows the two objects and the polygon faces where they are to be attached. The center points of the polygons are computed along with the normals to the polygons. Next, the normal direction for each polygon is set to the 'Z' axis. Vectors for the 'X' and 'Y' axes must also be constructed for each polygon. Two matrices are created that represent the positions and orientations for the point of attachment. The inverse of the matrix for the base object is multiplied by the matrix for the child object. This yields a matrix which describes the transformation of the child object in the base object's coordinate frame. This is the transformation on the child object necessary to line up the attach points. This matrix is stored in the child object's 'ref' matrix. Also, the child object's 'amat' is set to identity. This cancels any rotations or translations on the child object and forces the two objects to line up as specified. Rotations and translations can be done on a child object but will now be relative to the base object. The base object's 'kids' array is updated to show that the attaching object is now a child of the base object. The 'ref' matrix is put in the display list for the child object and a call to the child's display list is put at the end of the parent (base) object's display list.

Deleting an object would be a simple procedure were it not for the complexity introduced by attachments. The simplest method of handling this is deleting all children of an object that is deleted. However, this is not desirable. Therefore, when an object is deleted, all of its children are unattached and restored to normal status. One problem exists: child objects positions are defined by matrices that are relative to the parent object's position. Therefore, the child object's 'ref' matrix is not set back to identity, but is instead multiplied by the product of its parent's 'ref' and 'amat' matrices. This results in the object not moving from its current position in the world. One can think of this as a virtual object (invisible object) existing where the old parent object existed. This virtual object provides invisible support to the child objects, preventing them from collapsing inward.

Resizing an object is another procedure that would be simple if one did not have to deal with attached objects. After an object has been picked to be resized and the new parameters have been entered, a completely new object is created. If it is a child object, then its parent is looked for. The information regarding attachment points is stored in the parent. The polygons are the same as before except that the dimensions are different. New attachment points are calculated based on the new coordinates of the new object and the 'ref' matrix is calculated. The 'kids' array of the parent is modified to point to the new object and the old object is removed. If the resized object is itself a parent then the old object's 'kids' array is copied to the new object and all of the 'ref' matrices of the child objects are recalculated. Also, all references in display lists to the old object are changed to the new object and calls to any children are placed in the new display list. The old

object's display list is removed and the memory allocated to the object is freed.

When the link (or other structure) is complete, it is saved in a form able to be read by R2. R2 can not take ROBOSIM code and create editor structures from that. Therefore, one must save any files that might possibly be edited again. After all the links of a robot have been edited, ROBOSIM code can be generated. Currently, the editor is set up in such a way that after generating the ROBOSIM code, ROBOSIM is automatically called and the appropriate filename passed to it. ROBOSIM then generates the link files for the robot. If the user is on a terminal capable of using MD, then MD is automatically executed with the robot name passed to it. In this manner, it is much quicker and more flexible to use the editor, since the user does not have to exit the editor, run ROBOSIM, and then run MD.

R2 generates ROBOSIM code in a fairly straightforward, though not necessarily intuitive (especially when looking at the ROBOSIM code), way. The method used resulted from the difficulties involved in creating more "readable" ROBOSIM code. One method would have required a breadth-first traversal of the editor's hierarchical structures starting at the deepest level of the tree (the thickest part). Another method would have required more registers than ROBOSIM has if there were more than four child objects to any object. The method used can be thought of as resolving the hierarchical structure dependencies into a simple list. Remember that the position and orientation of an object affects all of its children objects by the fact that the children's position and orientation are described in the coordinate frame of the parent. All that has to be done is multiply all of the transformations down the tree and get one absolute transformation for each object. Then, the first object is created, moved and rotated, and then stored in register B. Each additional object is handled the same way except that register B is added to it and the result stored in register B. Once all the objects in a link have been processed, a "STORE-LINK" command is added. Each link is processed the same way until no more links are left.

Pictures 3.1 through 3.5 show two links of a robot being built. First, a cylinder is created and moved to one side. Then, a custom object is created. Next, the custom object is attached to the cylinder. The final steps for this link are a fixed joint attached to the base of the cylinder and a revolute joint attached to the custom object. Then, the link is saved. Another link, a simple box, is created. The input and output joints are made and attached to the link. Then, that link is saved. After these links have been saved, ROBOSIM code is generated for them and passed to ROBOSIM. The output from ROBOSIM can be seen, also. Now, the files describing these two links have been created and they can be looked at with MD. The ROBOSIM code generated for the base link (LOC link) is listed in Table 1. The structure of the link file generated by ROBOSIM is shown in Table 2.

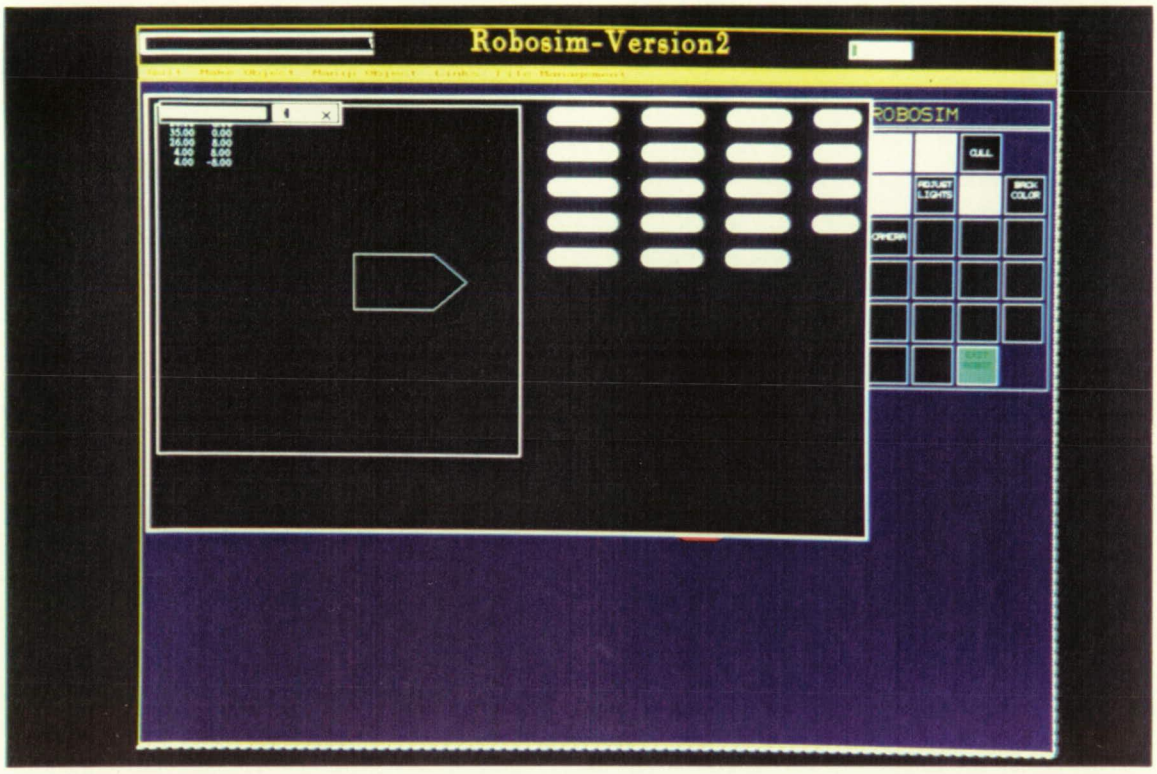

Figure 3.1: Creating a custom object

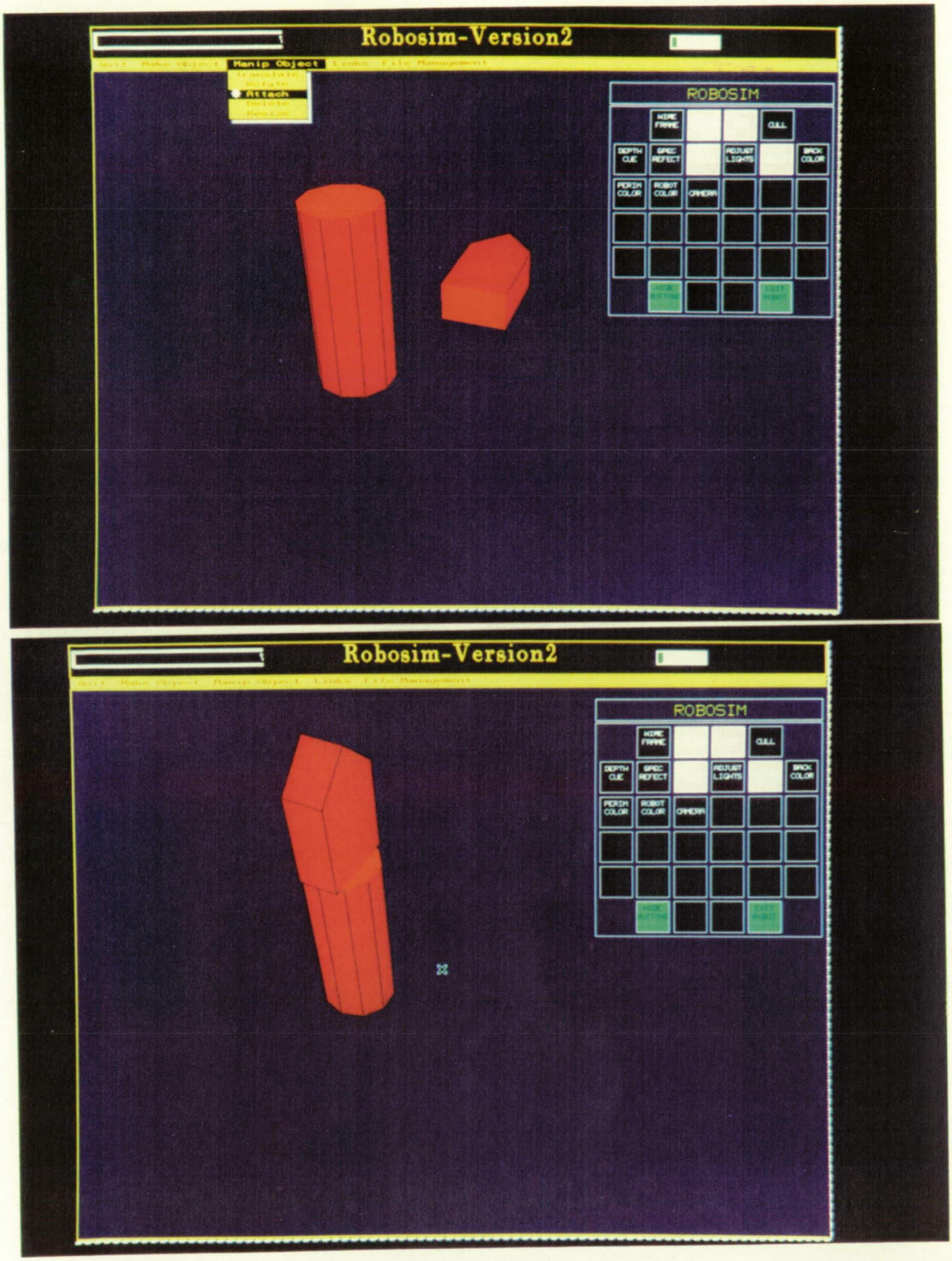

Figure 3.2: Cylinder and custom object before and after attachment

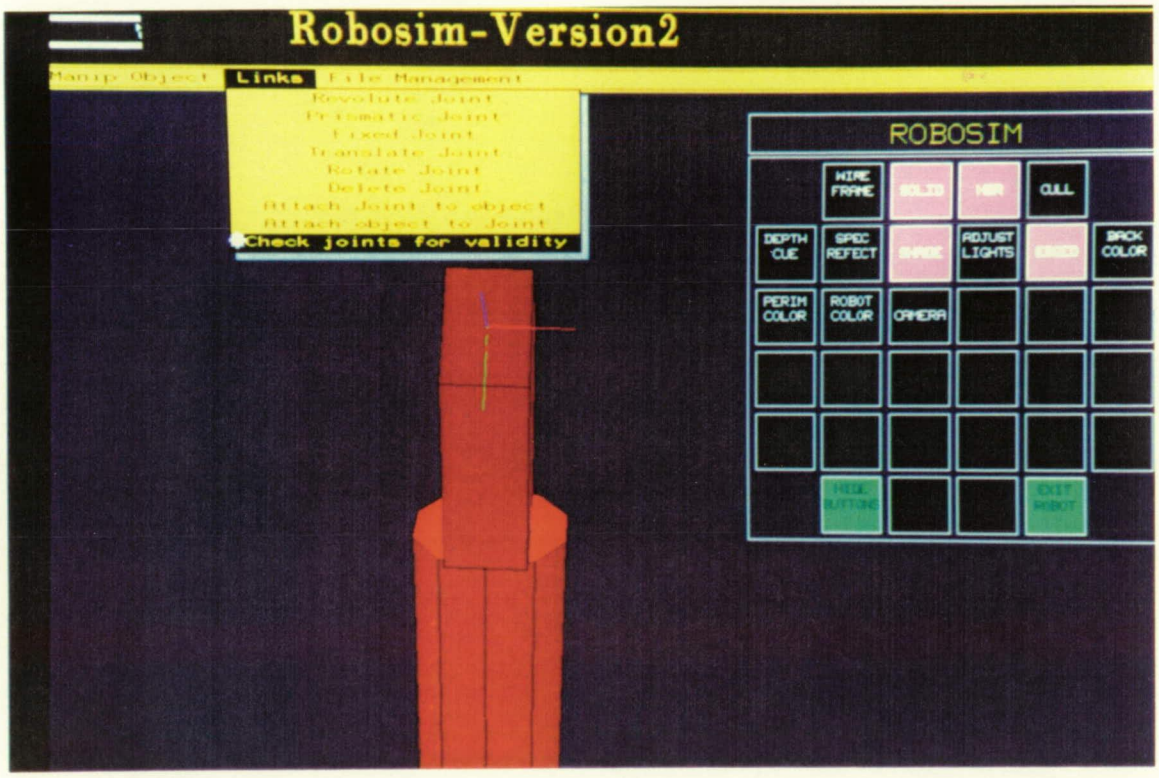

Figure 3.3: Base link with joints being checked

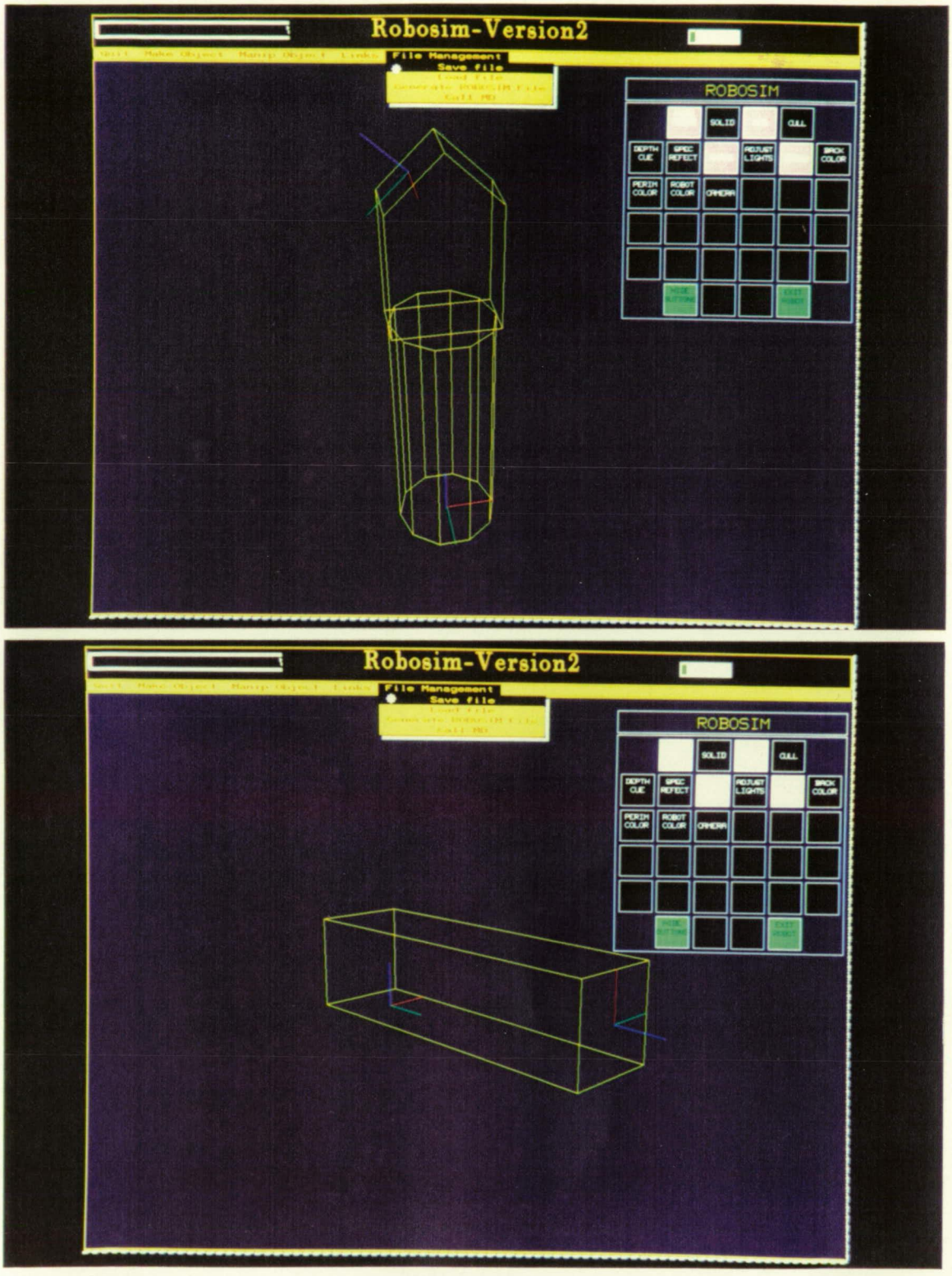

Figure 3.4: Base link being saved and compiled by ROBOSIM

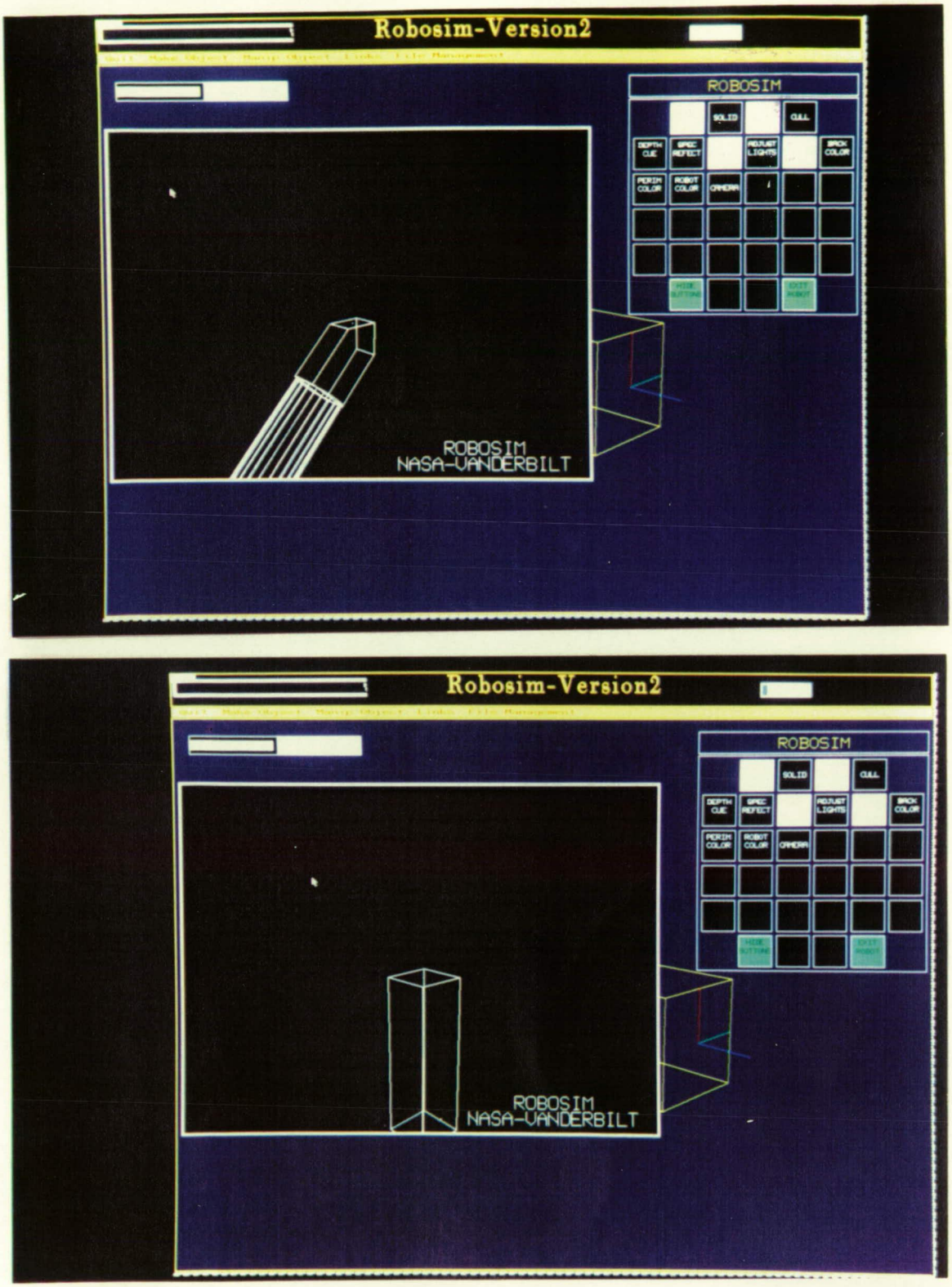

Figure 3.5: First link being saved and compiled by ROBOSIM

```
LOOK-FROM X=-100., Y=100., Z=45.
LOOK-AT X=0., Y=0., Z=8.
CLEAR
STORE B
R-JOINT-I+1
    ROTATE X=-45.000
    ROTATE Z=90.000
    TRANSLATE X=-5.000, Y=-30.000, Z=55.000
ADD B
STORE B
CLEAR

MOVE X=-10.000, Y=-10.000, Z=0.000
DRAW X=-10.000, Y=10.000, Z=0.000
DRAW X=15.000, Y=10.000, Z=0.000
DRAW X=25.000, Y=0.000, Z=0.000
DRAW X=15.000, Y=-10.000, Z=0.000
DRAW X=-10.000, Y=-10.000, Z=0.000
EXTRUDE-SURFACE Z=10.000
    ROTATE X=-90.000
    ROTATE Y=-90.000
    TRANSLATE X=0.000, Y=-30.000, Z=35.000
ADD B
STORE B
CLEAR

F-JOINT-I
    TRANSLATE X=0.000, Y=-30.000, Z=-25.000
ADD B
STORE B
CLEAR

CYLINDER R=10.000, H=50.000
    TRANSLATE X=0.000, Y=-30.000, Z=0.000
ADD B
STORE B
CLEAR

LOAD B
STORE-LINK C.LOC
VIEW
END
```

Table 1. ROBOSIM code generated by R2

Row	Col 1	Col 2	Col 3	Col 4
1	THETA	DZ	DA	ALPHA
2	JA1	JA2	JTYPE1	JTYPE2
3	AINERT (4X4)			
7	AJNT-I (4X4)			
11	AJNT-I+1 (4X4)			
15	AMAT (4X4)			
19	NVEC	UNUSED	UNUSED	UNUSED
20	X1	Y1	Z1	D1
21	X2	Y2	Z2	D2
:	:	:	:	:
NVEC+19	XNVEC	YNVEC	ZNVEC	DNVEC

Variable Definitions:

THETA	=	Denavit-Hartenberg parameter
DZ	=	Denavit-Hartenberg parameter
DA	=	Denavit-Hartenberg parameter
ALPHA	=	Denavit-Hartenberg parameter
JA1,JA2	=	joint defined flag
JTYPE-I,I+1	=	joint type -> Revolute,Prismatic,Fixed
AINERT	=	generalized link inertia
AJNT-I,I+1	=	transforms of input and output frames
AMAT	=	link's A-matrix
NVEC	=	number of vectors in list
Xi,Yi,Zi	=	x,y, and z component of vector
Di	=	move or draw vector

Table 2. Structure of Link File Created by ROBOSIM

3.4 Simulation Library and Environment

The simulation library and environment provides methods to access the data structures created by ROBOSIM. The robots and other objects are specified and loaded into memory. These structures remain resident in memory while the simulation is running. The library provides an interface to these structures so that the user does not have to understand what is happening at that level. The library provides higher level facilities much like an actual robot programming language.

The simulation package allows one to use the robots that have been designed. The package consists of a library of C functions that operate on the files created by ROBOSIM. Although this package is far from complete, it allows simple simulations to be run. Also, it provides a framework in which to test the major components for the simulator: collision detection and dynamics. Having the simulator be a library of C routines allows more flexible methods for running simulations. Very specific and efficient simulations can be written in C and which call the simulation functions directly. However, even at this level, much of the internal data structures is hidden from the user. This level of programming roughly corresponds to programming a robot in its programming language. For instance, one can tell a robot to move along a straight line or move a particular joint. A complete reference of simulation functions available can be found in Appendix B. Using these same routines a very flexible, user-friendly interface can be built up, allowing an interactive way to do simulations that are not too complicated, or that do not require great speed.

ROBOSIM provides most of the information required by the simulator by way of the files it creates. However, some information is not directly provided, but it can be determined from what is there. This involves the information required by the collision detection algorithm. ROBOSIM provides the Denavit-Hartenburg parameters, the A matrix, the pseudo-inertia matrix, and a list of points which describe the physical structure of the robot. The internal data structure also includes areas that are not currently used, but will be at a later time. These include minimum and maximum joint angles, velocities, and accelerations. The structure also includes information related to Starbase graphics. The actual C structure declarations used can be found in Appendix A. The simulation package acts as intermediary between the user and the internal representation.

The simulation program that the user writes can turn on collision detection, request solutions to inverse kinematics problems, and display results graphically. The user can use the general numerical Jacobian method for inverse kinematics or provide an exact solution for his robot. The user simply passes the address of the function to the simulator, and the simulator will then use that function when solving inverse kinematics for that robot. A proposed extension to the simulator will allow the recognition of the twenty-four possible robot configurations for which exact solutions exist. The exact solutions to these configurations would then be used instead of a numerical method, freeing the user from having to solve and code it himself. A good use of the simulation system can be found in a later case study section. This case study uses most of the features of the simulation system, as well as R2.

The simulation library's commands correspond to real robot programming commands found in many robot languages. Interfaces to many different robot languagues are planned. This will allow actual robots to be simulated, and then have a verified program downloaded to the robot. Additionally, the simulation could be run in parallel with the robot, with a planner or some other type of higher-level process sending the same commands to the simulation as well as to the actual robot. This can be used for verification, or even more importantly as part of a feedback loop to the planner. This will allow the planner to receive information from the simulation that it can not get from the actual robot. For instance, the simulation could provide forces and torques if the robot does not have sensors for that. Also, the planner could check out a plan of action on the simulation before actually driving the robot. This would let the simulation check for collisions or other dangers without risking the real robot.

3.5 Inverse Kinematics

The current default method for solving the inverse kinematics problem is the Newton-Raphson method. This method is an iterative method which uses the Jacobian of a robot. It is limited to six degree of freedom arms and has many other problems. The Jacobian is a six by six matrix that relates differential changes in joint angles to differential changes in world coordinate space. In other words, if you take the vector of joint velocities and premultiply it by the Jacobian the result will be the velocities of the end effector in coordinate space. Now, if you invert the Jacobian matrix, then you have a matrix that relates differential changes in coordinate space to differential changes in joint angles. Now, if the robot end effector is at a certain place and you want to know what joint variables would put it there then do the following procedure.

First, record the current joint angles. Then, compute the Jacobian and invert it. Now, subtrace the current position of the robot in coordinate space from the desired location in coordinate space. Multiply the inverse Jacobian by this difference. This yields a set of differences in the joint angles. Add this set of differences to the joint angles. Compute the new position of the end effector. Iterate this procedure until the error is acceptable low.

There are many problems with this procedure. First, due to singularities in the Jacobian, the method often does not converge. Second, when it does converge, you get only one possible solution and there is no way to get the others. Third, it is very slow. However, there are some robot configurations in which there is no exact solution, and therefore this is the only general way.

The implementation used does not yield very good results. However, it is faster than that used in the original ROBOSIM. This probably results from the use of LU decomposition instead of actually computing the inverse of the Jacobian. For example, if you are trying to solve the matrix equation (Y = JX) for X, one way would be to invert J and premultiply both sides by that. However, there is a faster way to solve this. J can be expressed as the product of two matrices, an upper diagonal matrix and a lower diagonal matrix. With J in this form, X can be solved by back substitution.

The best way to solve the inverse kinematics problem is to provide the exact solution. Although this is usually difficult, there are only 24 distinct configurations. This means that if a robot has an exact solution, its inverse solution can be expressed by one set of equations out of a possible 24. Currently, only one set of equations is implemented: the one corresponding to the PUMA 560. However, it is in a general form, in which six parameters (the lengths of the links) can vary. This method also allows one to get all possible solutions to the problem. In this way, additional constraints can be checked for, such as limitations of joints and checking different solutions to find one that does not collide with itself or other objects. This method is also faster by two orders of magnitude. Also, most commercially available robots have configurations that have exact solutions. It is not possible to run a simulation in real time using the numerical method, at least not without a floating point accelerator.

The only other method involves solving for five of the six joint angles analytically and using the Newton-Raphson method on one joint. This method will work on some configurations that do not have exact solutions. The usefulness of the method is better, yielding more solutions than the full Newton-Raphson method. In addition, the numerical part of the algorithm is not as sensitive to singularities, since it involves only one equation. This algorithm is not currently provided in the simulation, but the user could provide his own.

3.6 Collision Detection

Collision detection is very important in simulation of robots. One usually wants to know if the robot has collided with its environment or with itself. The following discussion does not delve into the theory behind the methods used, nor does it give an overview of collision detection. For a complete discussion of collision detection methods see Walter's dissertation from Cornell. The collision detection algorithm implemented here is very similar to the POCODA (POlygon COllision Detection Algorithm) algorithm given by Walter. The implementation used is given with special emphasis on those extensions to POCODA.

The algorithm used can be broken down into several subalgorithms. These will be discussed from lowest level to highest level. The assumptions used here is that all objects are defined by convex planar polygons. The problems involved in collision detection are as follows. Given a polygon and a point in the plane of the polygon determine whether that point is inside of the polygon. Given a polygon and a line segment determine whether the line segment crosses the plane of the polygon. Given two polygons determine whether they intersect. Given two objects determine whether they intersect. Given two bounding volumes around two objects determine whether they overlap.

The one equation to keep in mind throughout this discussion is the plane normal form of the plane equation.

$$\phi(P) = N.P + nd \tag{3.1}$$

Where N is normal to the plane, P is the point, and nd is the distance from plane to origin.

The plane described by this equation is the set of points P such that $\phi(P)$ is zero. Also, given N, nd, and a point P, the residue (ϕ) is zero if P is in the plane, positive if P is above the plane, and negative if P is below the plane.

The point-in-polygon problem is the most time-consuming operation. The method used to solve this problem is the reason why the polygons must be convex. The algorithm is to follow the polygon's edges around the polygon checking to see which side of each edge the point is on. If the point is to the same side of each edge then that point is inside of the polygon. This is checked by substituting the point into each edge's penalty function. The penalty function is a plane equation such that the edge lies in the plane and the plane is perpendicular to the plane of the polygon. The penalty function is calculated once for each edge and stored in the internal structure. See Appendix A for the C simulation structure.

$$pen(P) = M \cdot P + md \tag{3.2}$$

$$M = (N \times E)\|E\| \tag{3.3}$$

$$md = -M \cdot P1e \tag{3.4}$$

M is the normal vector to the penalty plane; it is the cross product of the normal to the polygon plane and the directed edge normalized with respect to the directed edge. md is the distance of the penalty plane from the origin. This penalty function can now be used to determine which side of an edge a point is on.

The algorithm for determining if a line segment crosses the plane of a polygon should be obvious from the above discussion. The two endpoints of the line segment are both substituted into the equation of the plane in which the polygon lies. If the residues of the two points are the same sign then both points lie on one side of the plane. Therefore, the line segment did not cross the plane. If, however, the residues have different signs, then the point at which the line segment crosses the plane must be determined so as to use it in the point-in-polygon algorithm. Given two points $P1$ and $P2$ which are the endpoints of a line segment and $\phi(P1)$ and $\phi(P2)$ which are the residues of $P1$ and $P2$ in the polygon plane, then the point along the line segment that intersects the polygon plane is Pc,

$$Pc = P1 + (P2 - P1) \cdot \phi(P1)(\phi(P1) - \phi(P2)) \tag{3.5}$$

Each object in a simulation is composed of polygons, but due to speed and efficiency requirements the above tests would be prohibitive. Therefore, some simpler tests are required which can quickly eliminate some objects from the more exhaustive tests. The method used is to perform tests on bounding boxes of the objects. A bounding box is described by a point and a vector. The point is the center of the box, and the vector is the half-diagonal vector of the box (i.e. it points from the center of the box to a corner). These values are determined by first determining the maximum and minimum values of the object along the x, y, and z axes. The center is calculated by averaging the maximum and minimum values along each axis. The half-diagonal vector is calculated by taking half

of the difference between the maximum and minimum along each axis. For instance, along the X axis:

$$Cx = (Xmax + Xmin)/2 \tag{3.6}$$

$$Dx = (Xmax - Xmin)/2 \tag{3.7}$$

Now, two bounding boxes overlap if the distances between the centers along every axis is less than the sum of the half-diagonal components along the corresponding axes. However, the two bounding boxes must be defined in the same coordinate frame. Typically, each object is defined in its own coordinate frame and has a transformation matrix describing the position and orientation of the object in the world coordinate frame. Therefore, a method is needed to transform a bounding box from one frame to the other. Given two bounding boxes, $B1 = (C1, D1)$ and $B2 = (C2, D2)$, and two transformations $T1$ and $T2$ which are 4x4 matrices describing position and orientation of boxes $B1$ and $B2$, respectively, let $C1$,$C2$ and $D1$,$D2$ be the center and half-diagonal vector of $B1$ in coordinate frame 2.

$$C1, 2 = [C1][T1]([T2] - 1) \tag{3.8}$$

$$D1, 2 = D1@[T1]([T2] - 1) \tag{3.9}$$

where @ is the dilation product, an operation between two matrices which can be expressed as the product of two matrices whose elements have all been changed to their absolute values.

In order to test for bounding box overlap given two boxes, one first has to express $B1$ in coordinate frame 2 and check for an overlap. Then convert $B2$ to coordinate frame 1 and check for an overlap. Only if both checks indicate an overlap is there one. If an overlap is indicated then further checks have to be made to determine if there is a collision.

Once a possible collision is indicated by overlap of bounding boxes, more exhaustive tests have to be performed. First, all points in one object must be transformed to the other object's coordinate frame. This can be done using Eq3.8 above where $C1$ is a point in object 1. Once this is done, a first approach would be to check every edge in each object against every polygon in the other object. However, there are some ways to reduce the number of edges which must be checked. First, each edge in object 1 is checked against the bounding box of object 2. Only if the edge falls within the bounding box could it intersect the object. Each edge that could intersect a polygon is saved in the reduced edge array. Now, each edge in the reduced edge array is checked against the polygons in object 2. However, each polygon from object 2 is first checked to see if the plane it lies in could intersect the bounding box of object 1. If it does not, there is no need to check edges against it. Finally, each possible edge is checked against each possible polygon, using the methods described above, to determine if a collision exists. If not, then all points of object 2 are transformed to the frame of object 1 and the procedure repeated. The following summary is from Walter's thesis.

- Compare the bounding boxes of each object.

(a) If the bounding boxes overlap then the likelihood of a collision is high and further checks are required, and the procedure continued.

(b) Otherwise the two objects cannot possibly collide. They may be declared collison-free, and the procedure is exited.

- The objects are transformed to a common reference frame by tranforming the points of j into the reference frame of k. The new object is referred to as (j, k).
- Edges in (j, k) are compared with the bounding box of k.

  (a) If an edge intersects the bounding box it is retained for further tests by inserting it into the reduced edge array.

  (b) Otherwise the edge is excluded from further tests.

- Check each polygon in k.

  (a) Check whether the polygon plane intersects with the bounding box of (j, k). It means comparing the polygon against all the reduced edges in (j, k).

    A. If the polygon intersects with an edge then a collision has occurred, and the procedure is exited with a collision condition.

    B. Otherwise, continue until all edges are considered.

  (b) Otherwise, the polygon cannot possibly be a source of collision, and is excluded from further tests between the two objects.

- Evaluate progress.

  (a) If this is the first time to this step then, interchange the roles of j and k and repeat all steps after (2), since a collision is still possible, although undetected this far.

  (b) Otherwise, the two objects do not collide. They may be declared collision-free, and the procedure exited.

This algorithm is the one used in the simulation library and environment with one difference. At step 3, Walter checks every edge in (j, k) against the bounding box of k. A much simpler first check is to check the plane of the polygon that contains the edge against the bounding box first. If that polygon does not intersect the box, then all the edges of that polygon are excluded. This is a much faster check than checking an edge against a box. This is similar to what is done for the k object in step 4.

ROBOSIM does not directly generate all the information required for collision detection. However, it can be calculated from what is provided, namely the vector list. The vector list is a list of points that define the polygons of the object. This vector list is split into separate polygons as it is read from a file. Then the normal and normal distance for each polygon is calculated and stored. Next, the penalty function for each edge is

calculated and stored. As the vector list is read in, the maximum and minimum x, y, and z values are saved and used to calculate the bounding box. The internal data structure now contains all of the information necessary for collision detection.

The use of this algorithm requires some special considerations when used with robots. The technique used employs a bounding box around each object in the environment, a bounding box around each link of each robot, and a bounding box around each robot. The bounding boxes around each object and each link are computed at load time, but the bounding boxes around robots must be computed as needed. This is because the bounding boxes around robots change as the joint angles in the robots change. Whenever a collision is checked for, bounding boxes are created around the robots. They are calculated by using the bounding boxes around the links. The minimum and maximum extents along the x, y, and z axes of the bounding boxes around the links are computed. Then a bounding box around all these bounding boxes is computed from the minimum and maximum extents. The purpose for bounding boxes around robots is that if there is more than one robot, even bounding box checks become expensive. If there are two robots, each with nine links (6 movable and 3 fixed), 81 bounding box checks would be required every time. And if there were three robots, 729 bounding box checks would be required. With three robots, and therefore three bounding boxes, only three bounding box checks are required. If there is a collision between two bounding boxes, only the two robots need be checked.

Previously, there was a transformation matrix associated with each link that described the coordinate frame of that link with the previous link. This is not adequate for collision detection, however. This matrix can be obtained by multiplying all of the matrices of the previous links together, yielding a transformation of the current link in the world coordinate frame. It is much simpler, and faster, to calculate this matrix for each link whenever joint variables are changed in the robot rather than waiting until needed by collision detection. This is especially true since collision detection checks are made from the end effector inward, as a collision is more likely with the end effector. Whenever a joint variable in a robot is changed by a library function, the transformation matrix of the link in the previous link's frame as well as the world frame is calculated and stored in the link's structure. Then, it is used by the collision detection algorithm as needed.

Another problem that requires special treatment is collisions involving the robot with itself. This is especially difficult when one considers that the design of the robot may include overlap of adjacent links. If this is the case, then if links of the robot are checked with other links of the same robot, then collisions might be seen that aren't really valid. Therefore, collisions are not checked for against adjacent links. The simulator has internal provisions for joint constraints. Therefore, any possible collision could be provided for by limiting the joint angles. However, given legal joint values, it is possible for non-adjacent links to collide. Therefore, collision detection of the robot with itself must be made. Given a nine link robot, 28 bounding box checks must be made to ensure no collisions with itself. However, this self-collision detection may be controlled separately (i.e. it can be turned on and off independently of the other collision detection), since the user may not require these tests.

Since many objects and robots may be loaded before they are actually used in the simulation, the collision detection uses the list that is created by the USE command. The USE command adds its argument to a linked list of objects, and inserts a call to it in the display list. Therefore, it will be displayed when the display list is traversed. Also, the collision detection uses the linked list of objects to check for collisions. If an object is not in use, the collision detection does not waste time checking it.

Once collision detection is turned on, checks for collisions are made any time the library functions are used to move a robot. If there is a collision then a collision structure is filled out. This structure returns pointers to the objects and link numbers if the objects are robots. The library functions pertaining to collision detection are included in Appendix B.

The collision detection algorithm has only two weak points. It does not handle concave polygons, and it will not signal a collision if one object is completely inside of another. The stipulation concerning concave polygons is not serious. ROBOSIM does not generate concave polygons unless they are the result of an custom object. Although R2 does not check for concave polygons, this feature could be implemented. In fact, algorithms exist to split concave polygons into convex polygons. Either of these features could be implemented fairly simply. The problem of not detecting a collision if one object is completely inside another derives from the fact the algorithm used is a polygonal collision detection algorithm and not a solid object one. However, assuming two objects start off outside of each other and movements are sufficiently small, then this should not prove to be a problem. This condition also prevents the ability of one object to pass through another (i.e. a movement is large enough that two objects do not overlap at any point). This algorithm does not detect collisions in the volume swept by an object moving between positions with another object, but rather only overlap of the objects at the starting and ending positions. But, if the distance between the positions is smaller than the smallest object, then there should be no problems.

The collision detection has been implemented very effectively. The low level collision routines require transforming points in one coordinate frame to the other. This requires multiplying all points by a transformation matrix. The Starbase graphics package provides routines to do this, as well as to multiply 4x4 matrices together. When there is a graphics accelerator in the system, Starbase uses it to do the calculations. This allows matrix multiplication as well as transformation of points to be done in hardware, which is much faster than in software.

3.7 Example: Surgical Positioner

Everything described up to this point has been tested, and is in use. R2 and the simulation package are being used presently to aid in designing a kinematic surgical positioner. Its application would be specifically for brain surgery. The idea behind it is this: the robot would not be capable of motion on its own. It would be attached to a surgical collar, and after calibration would be positioned by the surgeon, with joint encoders sending the

values of the joint angles to a computer. The computer would show the position of the robot superimposed on a CAT scan. In this way, a surgeon can quickly determine points of entry. Currently, this is accomplished by precomputing where the points would be and then determining them using the collar as a reference. Having a way to immediately see what the positions are would prove to be much more flexible. Additionally, a hollow tube could be attached to the end effector. With this, the robot could maintain a particular orientation while the surgeon takes a biopsy.

Current research is to determine whether a robot of sufficient accuracy can be built. ROBOSIM provides an excellent test bed to perform this development. R2 has been used to design the arm and specify the dimensions of the links. A basic configuration similar to that of the PUMA 560 has been used. Therefore, an exact inverse kinematics solution exists and is used. The simulation library and environment is used to test the arm. The tests include ability to reach all the required points on the head (without passing through it). The inverse kinematics equations generate eight different solutions. These solutions are checked using the collision detection algorithm to ensure that there exists at least one which will reach the desired position without touching the head.

An additional requirement is that the positional accuracy of this robot be small. However, the size and cost are also important factors, so the smallest joint encoders would be desirable. The relation of world positional accuracy to joint accuracy is fairly easy to determine. Given a joint encoder of a certain number of bits, the accuracy is the range divided by two to the number of bits. This gives an angular measure of the amount a joint encoder could be off. This is used with the Jacobian to determine the maximum positional error. The Jacobian relates differential changes in joint angles to differential changes in world coordinates. The error in position caused by each joint is first determined. Then, the sum of the errors is computed. This gives the maximum amount that the positioner could be off. (It assumes each joint is off in the direction to give maximum error.)

Once a robot is generated, the simulation can run without the user. All data is saved in a file for later analysis. The simulation can run without displaying any graphics, or the user can watch it as the robot is put through its paces. The part of the simulation written by the user is shown below. It is not a general type of simulation that would be applicable to a wide variety of problems. However, it is sufficiently general in that it encapsulates the requirements of the project, but it does so without being limiting. For instance, the requirements are that it reach certain points on a head (cylinder) without any part of the robot touching the head. The user cycles through the points that are required, and the simulation sends back information concerning whether the robot specified can reach the points without colliding with the head.

```
#include "sim.h"
#include <math.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0
```

```
int  i;
FILE *fopen(), *fpout;
int  puma_inv();

sim()
{
    ROBOT  r1=0;
    OBJ    o1=0;
    JOINT  array;
    float  J[6][6];
    float  angle;
    extern COLLISION S_CO;
    char   *filename="testout";
    float  m[4][4];
    float  CONV = M_PI/180.;
      /*
       * PRE does transformation along world axes
       */
      r1=GET_ROBOT("/users/robosim/source/manipulators/jo/models/T");
      PRETRANSLATE(r1,25.,0.,0.);
      USE(r1);
      o1 =
GET_OBJ("/users/robosim/source/manipulators/jo/models/HEAD.OBJ");
      PRETRANSLATE(o1,0.,0.,0.);
      USE(o1);
      C_SWITCH(TRUE);
      SET_INV(r1,puma_inv);
      /*
       * cover head in increments of 1cm over the length and 5 degrees
       * from 75 to 295 output results to file testout
       */
      fpout = fopen(filename,"w");
      set_joint_error();

      for (i= -20; i<21; i++) {
          for (angle=0.; angle<105.*CONV; angle+=5.*CONV) {
            get_location(angle,m);
            fprintf(fpout,"\n");
            fprintf(fpout,"%5.2f %5.2f %5.2f %5.2f %5.2f %5.2f\n",
                        lo[0],lo[1],lo[2],lo[3],lo[4],lo[5]);
            if(KINV(r1,m,array,TRUE)) {
```

```
                JACOB(array,J);
                error(J);
                if(!MOVEJI(r1,array,1)) {
                  printf("collision, y = %d, angle = %f\n",i,angle);
                  printf("link %d\n",S_CO.L1);
                }
            }
            else {
                fprintf(fpout,"point did not converge\n");
                printf("point did not converge\n");
            }
        }
    }
    fclose(fpout);
}

get_location(j,m)
    float j;
    float m[4][4];
{
      m[2][0] = -cos(j);
      m[2][2] = -sin(j);
      m[2][1] = 0.;
      m[0][0] = m[0][2] = 0.;
      m[0][1] = 1.;
      cross(m[2],m[0],m[1],1.);
      m[3][0] = (float) -20.1* m[2][0];
      m[3][2] = (float) -20.1* m[2][2];
      m[3][1] = (float) i;
      m[0][3] = m[1][3] = m[2][3] = 0.;
      m[3][3] = 1.;
      return(1);
}

float NUM_BITS[] = { 12.0, 12.0, 12.0, 12.0, 12.0, 12.0};
float single_joint_error[6];

set_joint_error()
{
      int i;
      for (i=0; i<6; i++) {
          single_joint_error[i] = 2.*M_PI/pow(2.0,NUM_BITS[i]);
```

```
      }
}

error(m)
    float m[6][6];
{
      int   i,j;
      float err[6];

      for(i=0; i<6; i++) {
           err[i] = 0.;
           for(j=0; j<6;j++) {
            err[i] += (float)fabs((double)m[i][j] *
single_joint_error[j]);
           }
      }
      fprintf(fpout,"dX = %f  dY = %f  dZ = %f",err[0],err[1],err[2]);
      fprintf(fpout,"  rX = %f  rY = %f  rZ = %f\n",err[3],err[4],err[5]);
      fprintf(fpout,"distance error %f\n",
          (float) sqrt((double)
err[0]*err[0]+err[1]*err[1]+err[2]*err[2]));
}
```

Currently, various configurations with twelve bit joint encoders are being investigated. It appears that twelve bit encoders will provide the necessary accuracy. The use of R2 and the ability to resize objects provide a simple means to quickly create a new configuration. The generalness of the simulation library allows the same simulation to be used with no modification. A detailed description of the simulation library commands is provided in Appendix B.

3.8 Port to the Intergraph Workstation

To make ROBOSIM available on as many platforms as possible, we have ported the basic ROBOSIM package and the simulation library to an Intergraph 3260 workstation. The Intergraph 3260 workstation is the high-performance version of the 360 model. It is based on the Clipper processor chip (which means it is code compatible with lower-grade models), and runs the CLIX operating system (which is a derivative of ATT Unix System V). It is e quipped with two graphics screens, which are suitable for applications requiring dual displays. The configuration received by us has a large amount of main and secondary memory, which makes it very appropriate for large-scale program development efforts.

The graphical programming interface is realized through various libraries which offer many facilities, including line and polygon drawing, shading, etc. The capabilities of the

libraries are compatible with those of the Starbase library on the HP machines. One notable difference is the lack of display list libraries on Intergraph.

The basic ROBOSIM modeling environment and the simulation library were ported to the Intergraph workstation. The porting involved two steps:

- the modification of the graphic library calls (because of the differences in the graphic libraries), and
- the substitution of the Starbase display list calls with appropriate modules (because of the lack of display list facilities).

The result of the port is a fully functional basic ROBOSIM package.

Chapter 4

Intelligent Graphic Modeling Environment

The ROBOSIM package, together with the enhancements described in the previous chapter, provides a powerful graphic tool for designing and simulating geometrical objects (including robots, of course) using an engineering workstation. But the real power of this approach can be utilized only by integrating the services of a graphic modeling toolkit with knowledge-based techniques. This chapter describes the first results of the ongoing research efforts to create such an integrated modeling environment.

First a critical review of the graphical modeling techniques of the basic ROBOSIM package is given, followed by the description of the system design and implementational considerations for an enhanced modeling and simulation package, called *Agent*, which was created on top of the ROBOSIM package. The simulation package is accessible through an object-oriented command interface, and it incorporates and extends the facilities of the basic ROBOSIM code, as well as those of the simulation library described earlier. One automation testbed facility (which also utilizes the MULTIGRAPH architecture) is the Hierarchical Description Language, the subsequent sections discuss the features of this declarative language, together with the interfacing techniques to the modeling environment.

4.1 Critique of the Basic Graphical Modeling Technique

The extensions to the ROBOSIM package described in the previous chapter greatly enhanced its capabilities in modeling different geometrical objects and systems. But we think that a graphics modeling environment should provide some additional features in order to fully utilize the potential of knowledge-based techniques in the graphic simulation of geometric systems. These additional features are summarized below:

- **Need for separate representation of objects:** Currently the ROBOSIM modeling environment does not support the separate representation of different graphic objects in its workspace. The *display lists* representing these objects are concatenated together every time a new object is added to the system. This makes the modification of complex objects very difficult, because the whole ROBOSIM command sequence creating the complex object must be re-executed whenever one of its parts is modified. This is especially a problem during the editing phase, since such operations are quite frequently needed here. The solution would be to maintain these objects separately – at least during the editing phase of the modeling. On the other hand, concatenating together the parts of a complex solid object would speed up the graphic simulation, so the desirable solution is to maintain both representation forms and use the appropriate one for each step of the modeling process.

- **Need for more graphics objects in the workspace:** A large graphics simulation program typically contains several independently moving objects. The programming model offered by ROBOSIM (graphic registers) limits the number of these objects - i.e. the complexity of the systems which can be modeled with it. The desirable solution is to allocate the graphic objects dynamically, which does not limit their number. Then each of these independent objects could be controlled separately during the simulation.

- **Multiple aspect object representation:** Many of the enhancements to the ROBOSIM package (collision detection, dynamics, etc..), described in the previous chapter are basically "add-on" packages to the original system, with separated data representation schemes. The system design could be made much more understandable if a central data base would be used, containing every aspect of each of the models stored in it.

The next section of this chapter describes the system design and the implementational considerations of the enhanced modeling package *Agent.*

4.2 System Design of the Graphic Simulation Environment

The system design architecture of the *Agent* package can be seen in Figure 4.1. The architecture is centered around a database which stores datastructures which contain information about:

- the geometrical properties of the entities of the world model,

- the part-whole relationships between the entities,

- the information necessary for displaying these entities (display lists),

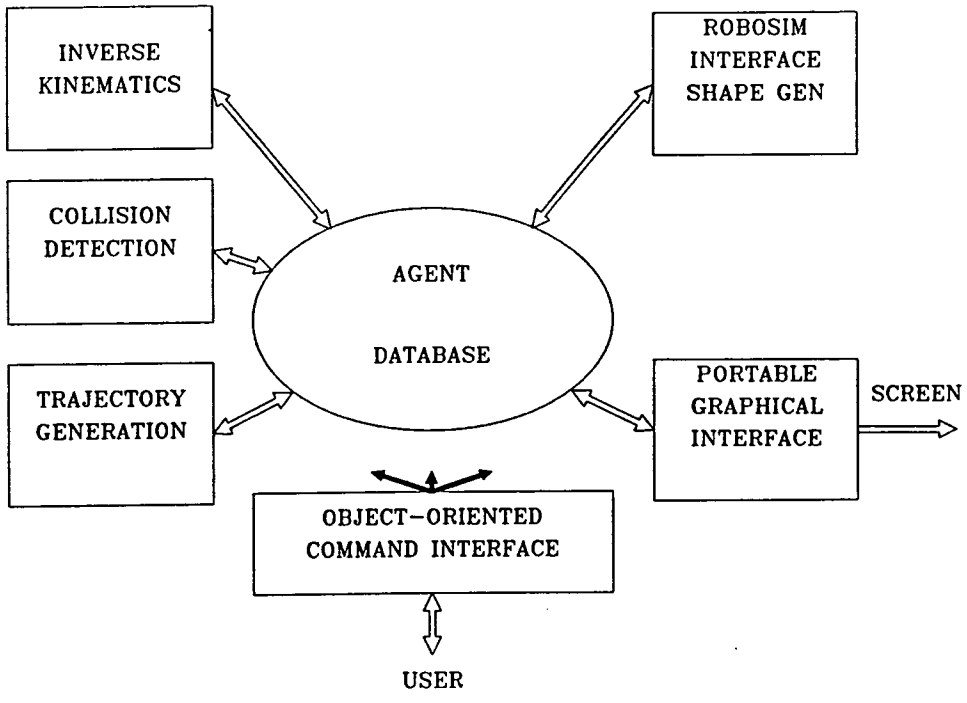

Figure 4.1: Main functional components of the simulation environment

- the information necessary for the collision detection algorithm,
- the information necessary for the forward and inverse kinematics simulation of the system (either in the form of data necessary for the default iterative methods, or in the form of analytical equations if these are available), and
- the information necessary for the forward and inverse dynamic simulation of the system.

Basically, the database contains all the information which was necessary to operate the models in the enhanced ROBOSIM package described in the previous chapter, but in a much better structured form. Unlike in ROBOSIM, where there was a limitation on the number of objects which can be handled by the system (fixed number of graphics registers), the objects in the database can be generated dynamically with no preset limit on their number or complexity.

In many cases the objects in the database are complex structures built of either less complex structures or elementary building blocks (like boxes, cylinders, spheres, etc..). Frequently there are objects having the same structure but with different parameters of their building blocks.

Structures (ie. robots) described in the ROBOSIM language can be instantiated in as many copies as it is required. This is made possible by the database which can also store

structural declarations of these complex objects which can be instantiated with the desired parameters whenever a new entity has to be generated. This way we can avoid having to build these objects from scratch.

The database is implemented as a set of data structures shared by the other active components shown in Figure 4.1. These active components, are the *functional blocks* of the system and they are based on the basic ROBOSIM package and the simulation library described earlier.

These blocks and their functionalities are as follows:

- **Object-oriented Command Interface:** Communicates with the user, interprets the user commands and builds and keeps track of the various objects.
- **Robosim Interface and Shape Generator:** Reads in and interprets files written using the (basic) Robosim language, and creates necessary objects using the Robosim shape generator facilities.
- **Inverse kinematics:** One of the basic robot simulation facilities. It solves the inverse kinematics equations, while the simulation is running. There are facilities for incorporating user-defined inverse kinematics routines (which solve the equations using analytical formulas and are typically very fast).
- **Collision detection:** Checks for and detects collisions, as it was described earlier.
- **Trajectory generator:** Generates joint angles dynamically, using various techniques (e.g. joint-interpolated motion, straight-line motion, etc.)
- **Portable graphical interface:** Interfaces the database to the graphical library (or hardware) available on the computer.

In the next section we describe the command interface to the package.

4.3 *Agent* command interface

The purpose of the *Agent* package is to act as a robot simulation environment which can receive commands from a user or from another program, for example an AI system's high-level planner. It provides features for environment configuration, manipulator control, and status reporting. The execution of the commands is performed in a graphics simulation environment.

The *Agent* is an interactive program, the commands entered by the user are immediately executed, and their results are printed or displayed on the screen. The program code is contained in one executable file typically named 'agent'. Typing 'agent' at the shell prompt will invoke it. Upon startup the agent initializes the graphics display and loads in some initial commands from the file named 'setup.cmd' if this file exists on the current directory. After this the agent's prompt appears on the screen indicating that it is ready to accept

commands from the user. The agent's input and output streams can be redirected to pipes, this way it is possible to issue commands by these programs.

4.3.1 Command format

The agent uses a character stream command protocol. Commands can be entered from the system console, loaded in from a file, or sent by another program using the pipe mechanism provided by the operating system. The general command format is:

```
[<label>:] <command-name> <argument1> ... <argumentN> [; comment]
```

with the following rules:

1. One command per line.
2. command parameters are separated by at least one white space (SP,TAB) character
3. There might be an optional semicolon at the end of the command, anything between this character and the end of the line will be considered as a comment. Lines beginning with a semicolon will be considered comment lines.
4. There is a way to create multiple line commands, by inserting a backslash character immediately before the line terminator character.
5. There is an optional label field in the command line. Labels are immediately followed by a colon character.
6. Movement commands can be issued only to the agents (robots) in the workspace.
7. Multiple agent movement commands per command line are possible. The individual commands are separated by commas in this case. Each of these should be directed to a different agent in the system, which will execute the commands parallelly. The execution of the next command line begins, when the last agent finished its operation. In contrast, if commands to two different agents are placed into consecutive command lines, the execution of these commands will be sequential.

4.3.2 Error reporting

While processing the environment configuration commands, the *Agent* generates an error log stream. For each command line which could not be processed, there will be an entry in this stream. The format of this entry is the following:

```
*** Error in line [label:NNN|NNN] --- <error code> <error message>
```

If there was a label preceding the command containing the error, then the error message will contain the name of the last seen label *and* the number of lines read since the last label was encountered. If the command stream did not contain labels, then the error message will contain the number of the command line counted from the beginning of the stream. (Line numbering is zero-based, that is the first line of a stream, or the line containing a label has offset 0.) Currently the following error messages are defined:

1 Undefined command: `<command>`

2 Unknown object: `<command>`

3 Too many arguments

4 Missing argument(s)

5 Illegal argument: `<index of command argument>`

6 Internal error (This error message will be given if the agent itself did not find any error in the command, but the execution of the command failed in ROBOSIM or in the simulation environment.)

7 Object not an agent: `<command>`

8 Multiple commands for the same agent: `<2nd. command for agent>`

9 Illegal coordinates: `<command>` (This message will be given if the desired coordinates are out of the agent's workspace, so no inverse kinematics solution exists.)

10 Joint violation: `<agent> <joint index>`

11 Collision: `<type> <object1> <object2> [<joint1> [<joint2>]]`, where type:

- **0** object to object
- **1** agent to object (1 joint field)
- **2** agent to agent (2 joint fields)

The optional joint fields are filled out in the error report if agents are involved in the collision.

12 Internal movement error. (This message is reserved for unforeseen execution errors within the Simulation Library.)

13 Hand is already holding object: `<object>`

14 Hand is not holding object: `<object>`

Note that in case the error message contains the original command line, only the offending command will be included from multiple agent commands.

4.3.3 Object creation commands

`make-object <object name> <object type> <parameters>`

This command creates an object in the workspace of the agent. Various object types have been defined, they and their parameters are described below.

`line <xend> <yend> <zend> [<color>]`

`polyline <x1> <y1> ... <xn> <yn> [<color>]`

`polygon <x1> <y1> ... <xn> <yn> [<color>]`

All lines objects are created with their starting point at the origin of the coordinate system. Polylines and polygons are always created in the XY plane, they can be rotated later if necessary. The optional color parameter is the name of a user defined color (see later). All other parameters are numbers. The agent does not make any assumptions about the physical units used, it is the responsibility of the user to specify the sizes of each object in a coherent way.

`box <xsize> <ysize> <zsize> [<color>]`

`cylinder <radius> <height> [<color>]`

`cone <radius> <height> [<color>]`

`truncated-cone <radius1> <radius2> <height> [<color>]`

`sphere <radius> [<color>]`

All solid objects are created with their center of mass at the origin of the coordinate system and their principal axis parallel with the Z axis. The color parameter is used in a manner identical to its usage at the line objects.

4.3.4 Object transformation commands

`translate-object <object> x=<xtran> y=<ytran> z=<ztran>`

`rotate-object <object> x=<xrot> y=<yrot> z=<zrot>`

These commands have a slightly different argument structure which serves the purpose of using defaults. If any coordinate direction is missing from the arguments, it is supposed to be 0. The order of the arguments is up to the user, i.e.:

```
translate-object box1 x=12 y=23 z=5
translate-object box1 y=23 z=5 x=12
```

are both accepted. Transformations are performed in the order of the arguments in the argument list. (This may make a difference in the case of rotations.) The rotate command expects its arguments in degrees.

4.3.5 Composite objects

`make-composite-object <name> <object1> <object2> .....`
`link-objects <name> <object1> <object2> .....`

These commands create a new composite object by joining the objects in the argument list permanently. They do not perform any transformations on the argument objects, but simply use their current positions. The first command will create a new object and leave the components in the workspace, while the second one will remove all components from the workspace after creating the composite object. Composite objects can be used (transformed, operated on by agents, etc..) in a manner identical to the elementary objects.

4.3.6 ROBOSIM objects

`make-robosim-object <name> <filename> <objectname> [<color>]`

This command can be used to create a composite object using its ROBOSIM source code model. The specified file is scanned until a `STORE-FILE` command is encountered with the specified object name as its argument. (See the ROBOSIM manual for more details.) As ROBOSIM does not have services for coloring objects, an optional color parameter is also accepted. It will be used to color the whole object. (I.e. regardless of the complexity of the ROBOSIM model, the whole object will be colored with the same color.)

4.3.7 Agents

`make-agent <name> <agent-type> [<color>]`

Creates an agent. Agents are basically robots whose models have been precompiled using ROBOSIM. Currently one example agent is available, the PUMA 560 manipulator. This agent model uses the real dimensions of the puma arm in millimeters, so size the other objects accordingly!

4.3.8 Object removal

`destroy-object <object>`

This command removes the descriptor of the named object from the agent's workspace. It might be useful when building composite objects and then selectively removing the unneeded parts.

4.3.9 Agent positioning

```
drive <agent> <joint angles>|<joint angle vector name>
drive-find <agent> <joint angles>|<joint angle vector name>
move-straight <agent> <coordinates>
move-inter <agent> <coordinates>
find-path  <agent> <coordinates>
move-straight-to <agent> <object> <coordinates>
move-inter-to <agent> <object> <coordinates>
find-path-to  <agent> <object> <coordinates>
```

Each of the above commands positions the agent. The coordinate specifications are agent specific, for the PUMA arms they must contain six values of either joint angles (only for the drive and drive-find commands) or rectangular coordinates (x, y ,z, roll, pitch, yaw) either workspace absolute or object relative (xxxx-to commands). The movement can be straight line (move-straight, move-straight-to) or joint-interpolated (drive, move-inter, move-inter-to), or the agent can be instructed to find a path to the desired location based on its knowledge of the workspace configuration (drive-find, find-path, find-path-to). The drive and drive-find commands also accept a previously recorded joint angle vector. (See later.)

```
translate-agent <agent> x=<xtran> y=<ytran> z=<ztran>
rotate-agent <agent> x=<xrot> y=<yrot> z=<zrot>
```

These commands are useful when small incremental motions of the robot arm are needed. They always perform straight-line motion. The coordinate specification uses the same scheme as the object transformation commands, that is the coordinates are named, and any unspecified coordinate direction is supposed to be 0.

```
drive2 <agent1> <agent1> <angles1>|<vector1> <angles2>|<vector2>
```

This is a version of the drive command for two manipulators. It is included only for backward compatibility, new programs should use the multiple commands per line feature. The parameter structure is identical to the single manipulator drive command.

```
movexy <agent> <x> <y> <z> <roll> <pitch> <yaw>
```

This command is useful for quickly positioning the agent. It will move the agent to the desired location (always interpreted as workspace absolute coordinates) in one step, without path synthesis and collision checking. It is useful for seeing target locations. This command will never operate a real robot manipulator, even if one is attached.

```
shift <agent> x=<xtran> y=<ytran> z=<ztran>
```

This is a single step version of the 'translate-agent' command. Its operation is similar to the 'movexy' command, and its parameter structure is identical to the 'translate-agent' command. NOTE: the 'movexy' and 'shift' commands cannot be used in multiple agent movement command lines!

```
minimal-step <value>
```

This command sets the robot movement simulation step size for the movement commands.

`set-solution <agent> <value>`

This command selects the inverse kinematics solution used for the agent. The current inverse kinematics method for the PUMA 560 arm provides 8 different solutions for (almost) any location. Some of these solutions are typically invalid due to joint angle constraints. The accepted range for 'value' is 0 ... 7. The argument to the 'set-solution' command sets the configuration the following way:

```
right handed:   0..3 (bit 2 = 0)
left handed:    4..7 (bit 2 = 1)

elbow down:     0,1,4,5  (bit 1 = 0)
elbow up:       2,3,6,7  (bit 1 = 1)

wrist down:     0,2,4,6  (bit 0 = 0)
wrist up:       1,3,5,7  (bit 0 = 1)
```

The command also accepts the special value of -1 which instructs the agent to select the most suitable solution automatically. This is the default operation of the agent. Note that setting a fixed configuration index will more likely result in joint limit violation error messages, since the agent has no chance for switching solutions. For small movements the automatic selection is based upon choosing a valid configuration which is closest to the current joint variable values. This strategy works best when the agent is performing various tasks in a relatively small part of the workspace. However for a major position shift this may not be the best approach. In such cases the agent will select a new configuration which offers the most room (i.e. all joint angles are as far from their respective limits as possible) for moving around in the vincinity of the new location. To determine which strategy tois to be used the agent compares the joint angles of the old and new positions. If the difference is larger than a preset threshold, then the second method is used, otherwise the first. This threshold value can be set with the following command (default value is 45 degrees):

`set-large-move-limit <limit>`

In most cases this strategy should work fine. However it is possible that in some cases explicit control of the robot arm configuration is necessary. (It is most likely to occur if relatively large straight-line motion segments are needed. In straight-line motion mode the agent considers a configuration change an error, since it would result in an abrupt reorganization of the links during the motion segment.) For such situations the agent offers the following configuration management commands:

`get-solution <agent>`

prints out the currently used configuration index. Note that you will get a value between 0 and 7 even if you use the automatic selection method.

`get-valid-solutions <agent>`

prints out the indices of all valid configurations for the current position.

`freeze-solution <agent>`

is equivalent to using 'set-solution' with the value obtained by using 'get-solution'.

The agent also performs collision testing while moving the robots in the workspace. The collision testing can be enabled/disabled with the following command:

`set-collision-check <flag>`

If the flag value is nonzero, collision checking is turned on, otherwise it is turned off. Initially the collision checking is enabled.

`display-tries <flag>`

This command can be used to enable/disable the display of the collision avoidance attempts performed by the agent (when using the 'find' family of the agent movement commands . If the flag value is nonzero, the avoidance attempts will be displayed, otherwise not. If the display is disabled, only the beginning phase of the operation (until the first collision is detected) and the complete collision free path (if found) will be shown. Initially the display of the avoidance attempts is enabled.

4.3.10 Position reporting

`get-position <agent>`
`get-angles <agent>`
`record-angles <agent> <joint vector name>`

These commands write a line of the following format to the report stream:

`position of agent <a> in line [label:NNN|NNN] --- <coords or angles>`

These commands behave differently based on the operating mode of the agent. If the agent is in simulation only mode, then the manipulator model's joint angles or coordinates are reported. If the agent is connected with a real robot manipulator then the robot hardware is queried for the actual joint angles, the agent's model is updated with the reported angles, and these angles are printed out. This way the usage of these commands will synchronize the agent's model with the actual manipulator. For the line numbering convention in the report stream see the explanation at the error report messages. The get-position command prints the position in world (rectangular) coordinates, while the get-angles command in joint angles (in degrees). The record-angles command is similar to the get-angles command, but it also records the angles in a coordinate vector which will be associated with the symbol specified in the command line. Coordinate vectors can be used as parameters for drive commands, and they are generally useful for recording important locations in the robot's workspace. The coordinate vector data base can be saved and restored with the following two commands:

`save-positions <filename>`
`load-positions <filename>`

As an additional safety measure against inadvertently losing important data, the agent automatically saves the current coordinate vector set into the last used file if either a save or a load command was executed previously.

4.3.11 Grasping

The agent's grasping operations are based on grasping attributes associated with each object in the workspace. These are the grasping coordinates – in an object relative coordinate frame – and the grasping opening used to establish contact with the object. The grasping attributes are best specified immediately after object creation, when its location is still known, and then the grasping coordinates will be transformed any time the object is moved. Commands:

`define-grasping-point <object> <x> <y> <z> <roll> <pitch> <yaw>`

establishes the grasping point and hand orientation in an object- relative coordinate frame.

`define-grasping-opening <object> <distance>`

establishes the hand opening which is used to grasp the object. The hand is CLOSED to the above distance upon grasping...

`define-default-grasping-gap <distance>`

This is the EXTRA hand opening above the value specified above when the hand is moving in to grasp the object. This is a global value, but may be overwritten for individual object by using...

`define-approach-opening <object> <distance>`

The default approach opening (= grasping opening + default gap) can be overwritten for individual objects using this call.

`move-to-grasp <agent> <object>`

This command moves the hand to the grasping point of the object and opens it to the approach opening (defined using either the default gap or the individual approach opening commands). It will give an error message if the hand already holds an object. This command uses the 'find-path' command's method to get to the desired point.

`grasp <agent> <object>`

This command grasps the selected object. The hand is already supposed to have been moved to the grasping point of the object, if not, an error message is generated. If the hand is not empty an error message is generated. Otherwise the hand closes to the grasping opening associated with the object, and in the simulation's data base a temporary link is set up between the object and the last link of the robot manipulator.

`release <agent> <object>`

This is the opposite of grasp. It gives an error message if the hand is not holding the specified object. NOTE: the agent does not model effects like gravity, etc.. If an object is released in the 'air' it will stay there in the simulated environment, but of course it will drop in the real world – leading to inconsistencies between the world and its model.

4.3.12 General graphics setup

`look-from <x> <y> <z>`
`look-at <x> <y> <z>`
`twist-camera <angle> <incremental>`
Establish parameters for the viewing transformation.
`define-color <name> <r> <g> <b>`
Specify color. Red Green and Blue intensities are 0.0 ... 1.0
`light-source <x> <y> <z> <color> <ambient>`
Specify light source. They are not needed for wire frame display, only for the other types. The ambient parameter just serves as a place holder, its value is not important. If it is present then the light source is ambient, otherwise it is directional.
`display-type wireframe|solid|shade`
Define graphics display option to be used.

4.3.13 General commands

`load <filename>`
Take commands from the specified file. Returns when end of file or the 'exit' (or 'end') command is encountered. Loads may be nested.
`set-echo <flag>`
Controls the echoing of the commands read from a load file. If the flag value is non-zero, echoing is enabled, otherwise it is disabled (it is the default upon startup).
`exit`
`end`
If given from the standard input exits the program. If given in a load file, finishes loading.
`clear`
Removes all object, agent, color, etc... declarations from the system.
This concludes the description of the command interface of the *Agent* package.

4.4 Automation Interface for Robot Modeling Systems

Previously we summarized the features we think are expected from a graphics modeling system to utilize the power of knowledge-based techniques in three dimensional world modeling. This section describes some of these knowledge-based techniques themselves. The Department of Electrical Engineering at Vanderbilt University has a long history of

building large knowledge-based engineering applications in the fields of instrumentation, process control, simulation and testing. In the course of this work we have developed numerous knowledge-based tools for this specific purpose.

The design of large-scale engineering systems that must operate in unstable, changing situations is one of the foremost challenges of the information sciences. Conventional design methodologies are based on the availability of a priori information about the environment and the system to be observed and controlled. The information is expressed in the form of models representing relevant aspects of the environment. The basic modeling principles of the system sciences such as *separation*, *selection*, and *model economy* are the key approaches for managing complexity. The essence of these principles is *simplification* until a model of manageable size is obtained. By imposing constraints on the possible behavior of the environment, the analysis and/or synthesis of the corresponding automation system becomes feasible.

There are two main ways how knowledge-based techniques can be used to satisfy the above goals. In many cases the more traditional *rule-based*, shallow modeling techniques can provide quite satisfactory results. The other approach is to use as much structural information about the environment as possible, in order to create a *structural*, deep model of the system. Both approaches have advantages over each other, so the best strategy is to use them together to solve complex engineering problems.

The graphics modeling toolkit described previously is intended to be used together with knowledge-based controllers. For the intelligent controllers using structural, deep modeling techniques, we use the MULTIGRAPH programming environment (developed at Vanderbilt) described later in this section.

We think that the two knowledge-based techniques can 'peacefully coexist' in complex systems using geometric, structural modeling. For example, in one possible application area, in Space Station automation, a typical scenario for the joint usage of the different techniques might be the following:

- Application areas for geometric modeling techniques:
  - The geometric model of the Station itself
  - Models of different manipulators operating on the outside or in the inside of the Station
  - Other moveable attachments to the Station, like solar panels, hatches, etc..
- Application areas for knowledge-based (rule-based) techniques:
  - Scheduling of different operations on the Station
  - Task Planning for robotics applications on the Station
  - Creating qualitative models of those subsystems which can not be modeled analytically due to their complexity or lack of information
- Application areas for structural modeling techniques (MULTIGRAPH):

- Modeling those subsystems where the structural and operational data is available to create qualitative, structural models
- Modeling control systems
- Fault propagation modeling and failure analysis

Of the above three techniques, the geometrical structural modeling toolkit has already been described in this report, and the rule-based techniques are supposed to be well-known, since they have been in use for quite a long time. But we think, that the structural knowledge-based modeling methodology and its run-time environment (the MULTIGRAPH architecture, which has been developed at Vanderbilt), deserves some more explanation.

Model-based knowledge-based methodologies have great potential in implementing automation systems for a wide range of applications. The main idea is quite straightforward and includes the following steps.

- A dynamic model of the environment (the system to be observed or controlled) is included in the higher-level knowledge-based controller of the automation system.
- The model is continuously updated based on observations.
- The control system is modified (structure and parameters) if state changes in the model require it.

We will focus on the computational problems of creating structurally adaptive controllers by using model-based techniques. The purpose of the discussion is to show the key components of a programming and execution environment that can be used for implementing this new system category.

The main computational requirements in the implementation of structurally adaptive controllers are the followings:

- The dynamic model of the environment and its interactions with the structure of the control system must be represented.
- The representation must be used as part of the control process, i.e. changes in the environment model must be mapped into changes in the structure of selected automation system components.
- The structural changes must be executed without suspending the system operation.

By using artificial intelligence terminology, the first requirement creates a *knowledge representation* problem. Naturally, the model-based approach demands the explicit representation of automation system models. The key issue is what kind of representation techniques can be used for this purpose? The second requirement addresses the problem of *knowledge utilization.* The knowledge which represents the interactions between the environment and the structure of the control system has to be actively used for modifying the

system operation. The problem is how to "convert" this knowledge dynamically into implementation specific terms? The third requirement is closely related to the *computational model* used in the execution environment of the control system. The question is what kind of computational model can support the dynamic reconfiguration of a processing system in execution time?

The main difficulty in the technology of intelligent adaptive automation systems is that realistic implementation can not be built without finding satisfactory solution for each of these problems. In the followings we will focus on the description of the components of the Multigraph Architecture which has been designed to serve as a generic programming and execution environment for this system category.

The Multigraph Architecture (MA) has been developed for building a broad category of intelligent systems operating in real-time environment. The MA has been used as a framework for intelligent instrumentation, automatic test configuration, and process control systems. The basic layers of the MA are the: (1) hardware layer, (2) system layer, (3) module layer, and (4) knowledge layer (Fig 4.2). In Fig4.3, the three main levels of the MA are shown from the user's point of view.

- **Model Designer.** The design and implementation of model-based, intelligent control systems requires extensive modeling. Because the unforeseen operational conditions might require structural modifications in the control system, the models must be hybrid. Hybrid models explicitly represent not only quantitative, but qualitative, structural attributes of the environment and the control system. Model designers must be supported by appropriate tools to build and validate these models.

- **Application Programmer.** The models that are used in the design and implementation of intelligent automation system are domain specific by their very nature. The form of the models (concepts, relationships) are different in chemical processes, mechanical processes, information processing systems etc., because the models must reflect the selected properties of these systems. However, some of the basic modeling principles, such as composition techniques, organization in levels of abstraction, multiple-aspect representation, etc. are quite universal. This generality makes it possible that the creation of domain specific modeling tools can be supported by general methodologies. The application programmer level in MA includes those components that are used for building various, domain specific modeling environments.

- **System Programmer.** The lowest level of MA provides interfaces to the components of the Multigraph Execution Environment (MEE). The central element of MEE is the Multigraph Kernel (MK), which is the run-time support of the Multigraph Computational Model (MCM). MCM is a macro-dataflow model which satisfies the required dynamic behavior mentioned before.

The models that are created during the modeling process are complex structures representing different aspects of the environment, the control system and their interactions.

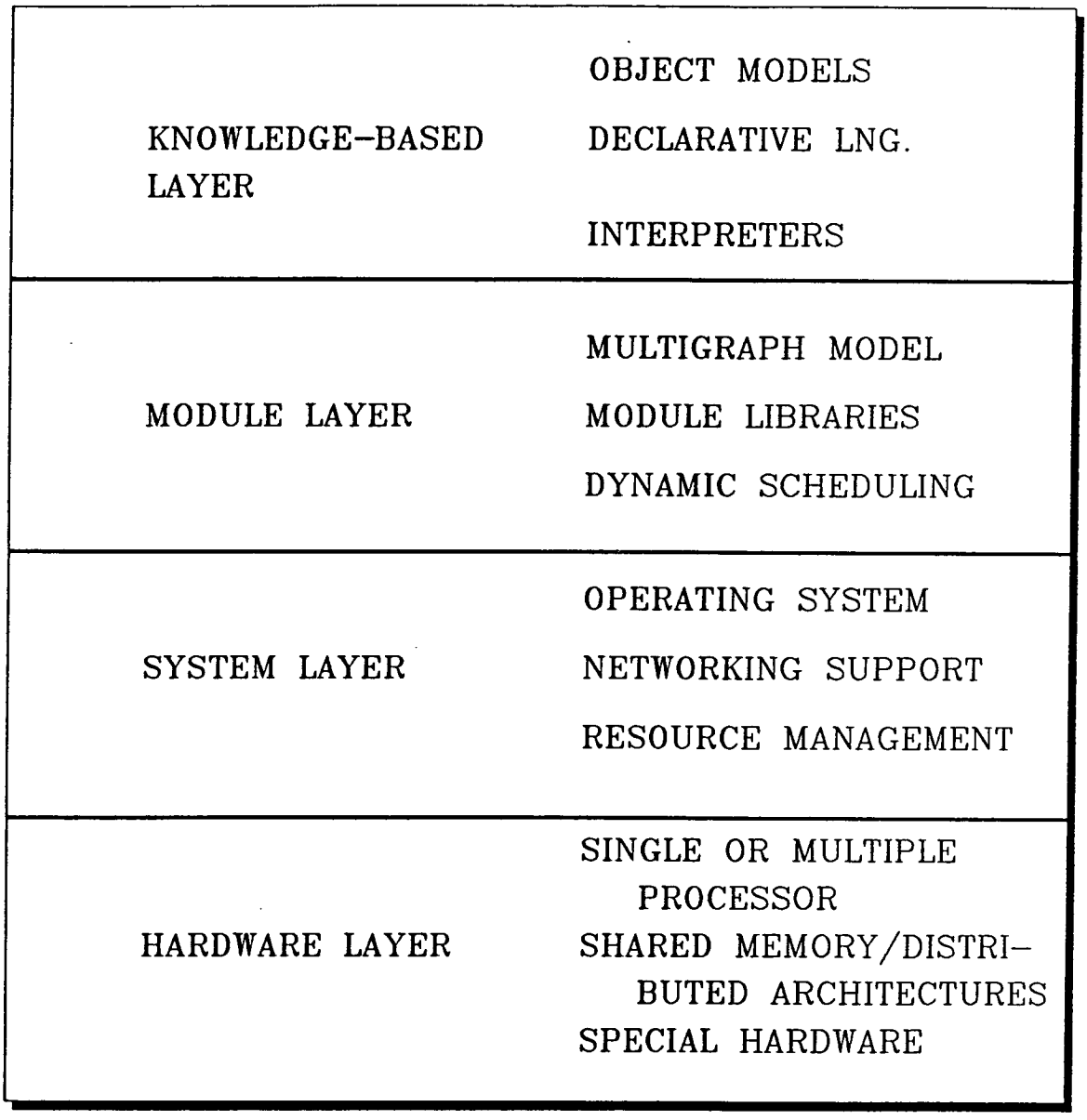

Figure 4.2: Layers of the Multigraph Architecture

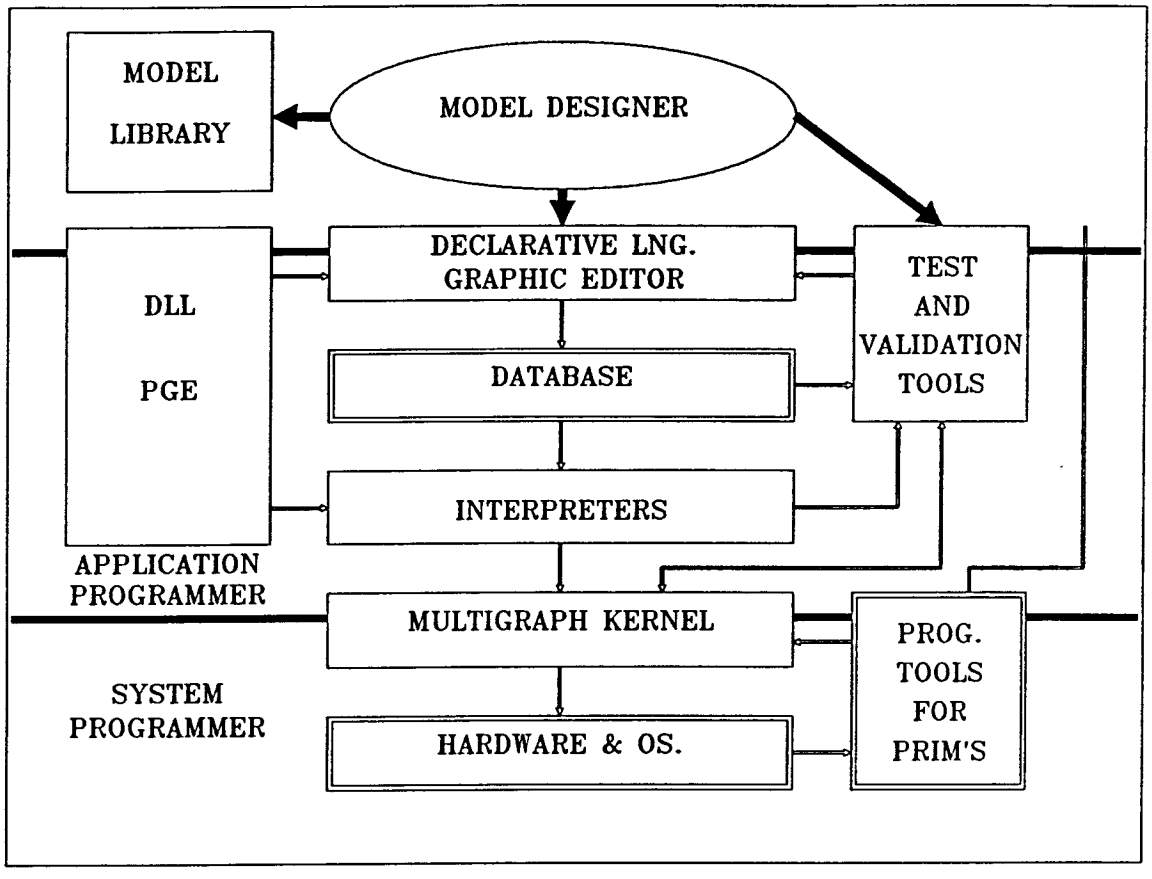

Figure 4.3: Structure of the Multigraph Architecture

It is important to note that in these models the structural complexity is the dominant factor, the algorithmic complexity is typically negligible. This fact had deep influence on the properties of the Multigraph Programming Environment (MPE). The two basic techniques used for supporting this activity are (1) multiple-aspect model building and (2) declarative/graphic programming.

- **Multiple-aspect model building.** Characterization of objects from different aspects is a well known method in modeling. There are artificial intelligence (AI) tools that directly support the creation of "multiple views". According to our experiences, the real difficulty is not the representation of different aspects but the expression of the interactions among them. The critical question is how to facilitate the well structured representation of these interactions? MPE allows the declaration of *structurally independent* (SI) and *structurally dependent* (SD) modeling aspects.

- **Declarative/graphic model building tools.** Modeling requires tools for representing the models. The representation technique has to satisfy two contradictory requirements. First, the representation system must provide "interface" for the model designer, i.e. the represented model has to be easily comprehensible by humans. Second, the represented model has to be machine readable, because the models constitute the "knowledge-base" which determines the system operation. Based on these requirements and on the fact that the models express dominantly structural information, MPE supports two equivalent representation form: **declarative languages** and the corresponding **graphic representation**. The model building process, which is performed by the model designers is fully graphical and directly supports SD and SI modeling.

  Figure 4.4 shows the graphic model a reconfigurable controller for a simple robot arm. The arm is controlled by (a) a proportional controller, or (b) a PID controller. The reconfiguration occurs when the "Checker" finds the performance of one of the controller unacceptable. The figure shows only the top level structure of the controllers and the simulation model of the arm. Each of the boxes have an internal structure on the lower levels of the hierarchy. The graphic model has been built by using the iconic editor of MPE. There exist an equivalent declarative language representation of the model. This declarative language is a variation of the "frame languages", which can be easily defined for the different modeling domains.

- **Test and Validation Tools.** Declarative languages offer excellent opportunity for automatic test and validation. The basic approach used in the test and validation toolset of MPE includes the following steps:

  - the declarative language forms are mapped into a unified graph structure,
  - test and validation criteria are defined for the different modeling aspects,
  - the criteria are expressed as graph properties, and
  - graph algorithms are used to check the properties.

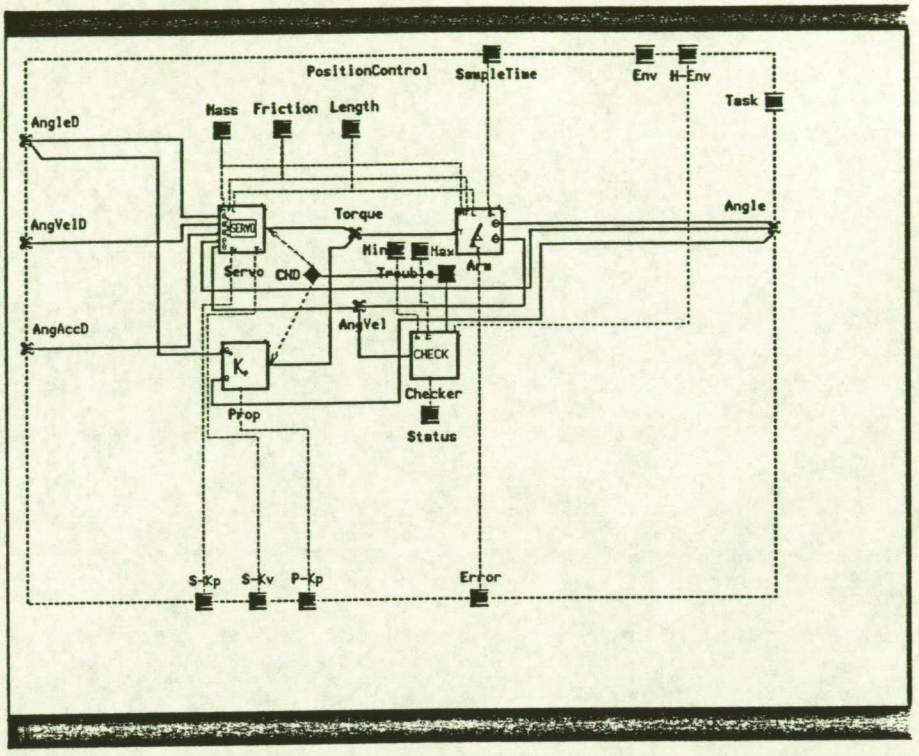

Figure 4.4: Graphical model for a reconfigurable controller

The methodology supports the automatic consistency testing of the individual modeling aspects and the consistency testing among the SD aspects. A serious limitation of the test approach is that only static properties of the models can be tested this way. In a new research direction we address the problem of testing the dynamic, run-time behavior of the system.

An important goal of MPE is to facilitate the definition of declarative languages and the corresponding graphic editors for new application domains. Generic tools belonging to the level of the Application Programmer support this task which includes the following steps: (1) definition of the syntax of the declarative languages, and (2) configuration of the corresponding graphic editor. The two programming tools developed for this purpose are the Declarative Language Language (DLL), and the Programmable Graphic Editor (PGE), respectively.

Multiple-aspect models of the external environment (platforms, signal sources, etc.), the various components of the control system (monitoring systems, controllers, etc.), and

their interrelationships embody the information that is necessary to generate a specific **instance** of the knowledge-based controller for the automation system. The problem of system integration is to generate this instance from the models, or in other words, to map the models into an appropriate executable program. Because of the implementation method of this mapping, we will call this process **model interpretation.**

The complexity of the model interpretation process largely depends on the nature of the models. If it includes only the symbolic, static model of a specific system, e.g. the model of a controller, the model interpretation process is reduced to the complexity of simple application generator systems. In the general case, the structurally adaptive controllers require the following capabilities from the model interpretation process.

- **Multiple-aspect interpretation.** The result of the model interpretation process must generate more than one subsystems. Multiple-aspect model interpretation means that the mapping process must interpret the models **from the aspects of the various subsystems** to be generated.

- **Decision making.** The complexity of the mapping process is largely the consequence of the fact that the models are not structured according to the subsystems of the system to be generated. (Except the simple application generator problems, where modeling is usually constrained to specific computation systems to be generated.) Indeed, in model building time the natural way of thinking is to focus on selected aspects of the environment, the control system and their interactions **without** any explicit considerations to the actual way of implementation. The model interpretation process has to be "smart enough" (1) to collect the relevant information from the models for the various subsystems, and (2) during this process to make decisions on the actual structure of the computation system by analyzing the interaction of the different modeling aspects.

- **Dynamic behavior.** The essence of any structurally adaptive system is the capability for dynamic reconfiguration of subsystems after a change in the working environment has been detected. It means that the model interpretation process has to be restartable from that point which has been effected by the detected change.

These capabilities required the elaboration of a special computation model in the Multigraph Execution Environment (MEE). MEE provides a system integration tool by supporting the dynamic configuration of application programs from a library of precompiled elementary processing modules. This configuration process can be performed by the higher-level knowledge-based system components using an appropriate builder interface of the MEE. Frequently the usage of the MEE also enables the utilization of the inherent structural parallelism in the application programs, since it is quite typical that many of the processing modules of an application configured using the above method can be executed concurrently, provided that the underlying hardware architecture supports this.

MEE uses a macro-dataflow model as its basic computational model. The reasons for this choice were (1) the well-known nature of the dataflow computations due to the

significant amount of research conducted on exploring the theoretical properties and implementational issues of these, and (2) the fact that many engineering system models (for example the signal flow graphs used in signal processing and process control systems) can easily be mapped into dataflow graphs. Some extensions were added to the "typical" dataflow computational concepts, because the MEE serves as a unified run-time support for the different parts of the intelligent automation systems, and these parts might use different models of computation (for example signal-flow graphs, discrete event simulators, rule interpreters, constraint propagation networks, etc..).

The applications in the MEE are mapped into a **control graph**. A control graph in the MEE is defined by its **actornodes, datanodes** and **connection specifications**. The actornodes are the active components of the graphs. They execute an application module (the script) which can be written either in Lisp or in other non-symbolic languages (C, Fortran, Pascal). The scripts are position independent, they communicate with the other graph components using the communication primitives of the MEE and the ports attached to the actor node. If the code of the script is reentrant, it can be attached to several actornodes. The MEE provides a way to pass a local parameter structure to the scripts, which is called the context of the actornode. Beside the typical dataflow control principle (a node can be fired whenever all of its inputs are present - **ifall** mode) MEE also supports another mode of actornode execution, where a single input data is enough to fire a node (**ifany** triggering mode).

The datanodes are the passive components of the control graphs. Their function is to store the data generated by the actornodes. They can store either a stream of data, or only the last data sent to them.

MEE supports several operation modes of a control graph. A graph can be operated either in **data-driven** or **demand-driven** mode, or in a combination of the two modes. In the data-driven mode, the data sent to a datanode propagates a control token to the following actornodes, which will fire after collecting the necessary tokens. The demand-driven mode means that an attempted read operation on an empty datanode will send a request token to all possible sources (ie. the connected actornodes) of the information.

MEE provides an environment and task structure which is used to assign the various system resources of the system hardware and software (processors, tasks, special hardware units, etc..) to the execution of the actornodes in the computational graphs.

The structure of a typical implementation of the MEE can be seen in Fig4.5. MEE can be depicted as a set of protected data structures which can be accessed through the following three interfaces:

- **Module Interface:** which provides the data and request propagation calls for the application modules attached to the actornodes.
- **System Interface:** which is responsible for scheduling the elementary computations using the system resources provided by the host operating system.
- **Builder and Control Interface:** which provides the control graph building and execution control facilities for the higher-level knowledge-based system components.

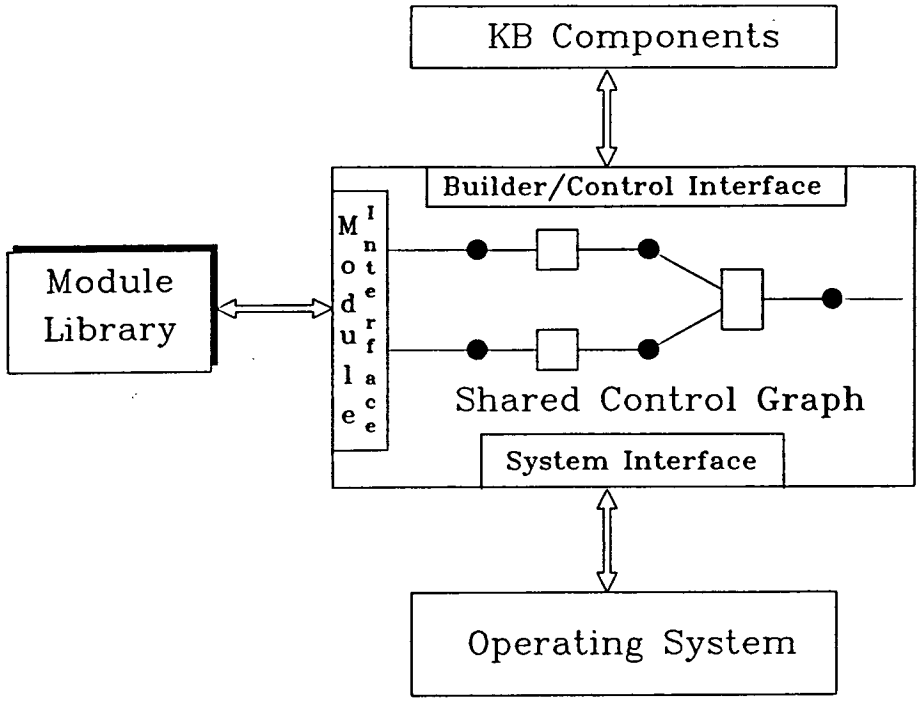

Figure 4.5: Structure of the MEE

The services of this interface can operate on an already active computational graph, which enables the **dynamic reconfiguration** of the application programs.

MEE offers a set of debugging tools which are especially helpful in concurrent systems. These include a stepper/tracer facility and a graphic monitor, which generates and displays the graphic layout of selected parts of the control graph, and dynamically displays the status of the nodes in the graphic window.

The computational model and the details of its implementation were selected such that the Kernel can provide the same execution environment on a variety of computer architectures, by hiding the details of the (possibly parallel) execution from the application modules, which can be simple sequential procedures in every case.

4.5 The HDL System

The Hierarchical Description Language (HDL), described below, is an important component of the automation testbed. Robot simulation should involve simulation of the *robot control system*, and this is the place where HDL plays the role of the knowledge-based component. It supports the definition of signal processing systems using very high-level facilities. These facilities include a graphical editor for editing HDL declarations (i.e. block diagrams). The HDL system is embedded in a Lisp environment which can communicate with the *Agent* system described above.

4.5.1 Introduction

This system has been developed for the generation and dynamic modification of real-time signal processing systems. It includes: (1) a declarative language, **HDL** for the hierarchical representation of procedural networks, (2) an interpreter which builds a specific version of the signal processing system, and (3) an execution environment. The execution environment is provided by the Multigraph Kernel (MK), which is the run-time support of the Multigraph Computational Model (MCM).

The system works like a hierarchical, planner in the following sense:

- the signal processing system (or more generally, a procedural network) is built according to a set of specifications, which is defined as goal for the interpreter,
- the "selection rules" driven by the required specifications are structured hierarchically,
- the building process can be contingent upon events detected in the execution environment,
- events can be fed back to the building process, and may initiate the modification of the existing signal processing system.

The outcome of the planning process is a signal processing system (procedural network) which runs under the control of the Multigraph Kernel. The run–time environment can be parallel, the network may run on a single– or tightly–coupled multiprocessor configurations. The implementation also supports the distributed computing environments, i.e. large procedural networks can be generated for computer networks. The building process generates a dynamic user interface and execution monitor for the network, which can be used for controlling and monitoring the system operation. A graphic monitoring facility is also available for displaying the hierarchical structure definitions.

This manual was written for those, who are already familiar with the Multigraph Computational Model, the basic representational principle used in HDL [1], and who have experiences with the Multigraph Kernel [2].

First, the semantics of the HDL is described, then the monitor and the user interface are discussed. The last section contains an example program. In the Appendix C the "C" interface of the system is described.

4.5.2 Semantics of HDL

The representation scheme used in HDL is described in [1]. The basic concept of the language is the module. A module can be either primitive or compound. Primitive modules do not have internal components, compound modules consist of other (primitive or compound) components. It is obvious that the primitives are related to actornodes executing a primitive algorithm while the compounds can mean a group of actornodes with internal connections. The module descriptions (in the sense of the language) are only structural specifications i.e. definitions and declarations, and do not include directly the code of the primitives.

An important feature of HDL is the parametrization of the structures. The parameters are entities which determine some properties of the structure. Parameters can be either *static* or *dynamic*. The static parameters are fixed during the interpretation of the structure and there is no way to change them without generating of a new structure. The dynamic parameters can be changed dynamically, and can be used for controlling the operation of the signal processing system. (This is why we usually use the term "control parameters" for the dynamic parameters.) A typical dynamic parameter is the "Start-Stop-Signal" for a module, or a the "tunable" time-constant of a digital filter, etc. Every block in HDL can have both static and dynamic parameters. The parameters are not constrained, which means that it is the users responsibility to handle and to interpret them in the scripts consistently. Parameters can be defined in compound strucures and they can be passed along to other component structures. by the low-level primitives written presumbaly in non-symbolic languages. The parameter mechanism is built on the context mechanism of the Multigraph Kernel in the sense, that a parameter set for a particular primitive is the context of the corresponding actornode.

4.5.3 Declaration of Primitive Modules

A primitive can be defined with the declaration :

```
(defprimitive
 Name Discipline
 Io_interface Parameter_interface Control_interface
 Environment_interface Task_interface
 Icon Picture
 body )
```

The `name` identifies the primitive. The `Discipline` determines the control discipline of the corresponding actornode, it can have one of the values `:ifall` and `:ifany`.

The `Io_interface` is a specification of signals used and driven by the primitive. It has the following syntax :

```
(Input_signal_list -> Output_signal_list)
```

The signals correspond to the datanodes to which this actornode will be connected. Either of the signal lists can be empty which means the lack of the corresponding connections. If it is not empty it ' contains elements of the form:

```
Signal_name
```

or

```
(Signal_name Signal_type)
```

The former denotes a signal of type "Stream" while the other one denotes a signal with the specified type (either `:stream` or `:scalar`). A run-time checking procedure will test whether the passed signals have the same kind as it was specified in the specification of the primitive. Identification of the signals is done by position.

The `Parameter_interface` consists of a list of parameters:

```
(Parameter_name*)
```

The parameters are the "static" parameters for the primitive, the `Parameter_name` is only a symbol specifying the name for the parameter.

The `Control_interface` defines the dynamic parameters for the the primitive, and has essentially the same syntax as the parameter interface:

```
(Control_parameter_name*)
```

The control parameters are the 'dynamic' parameters for the primitive.

From the static and dynamic parameters a parameter table will be constructed which will constitute the context of the actornode. The parameter table is a vector (in the case of actornodes running Lisp scripts) or an MGK buffer (in the case of foreign actornodes).

The slots are filled according to the sequence order of the static and dynamic parameters: first come the static parameters, then the dynamic parameters. In the case of the static parameters the values are copied into the table, while in the case of the control parameters a pointer to the value is passed. (It is necessary because the user might want to modify the value of the dynamic (control) parameter and the actornode should sense the change in some way, too.)

The `Environment_interface` and `Task_interface` are lists containing exactly one symbol which identifies the environment and task in which the primitive is executed. They are here merely for compatibility with the compound declarations, they do not play any role.

The `Icon` and `Picture` components are solely used by the graphics editor, their contents should not be the user's concern.

The `Body` of the primitive can be either a string or an arbitrary Lisp function body. If it is a string it is interpreted as the name of the foreign module to be handled by the Multigraph Kernel. If it is a Lisp function body (i.e. an implicit `progn`) a new function will be created which has the name of the primitive with the string "-fnc" concatenated. The body of the Lisp function will contain the definition as specified in the primitive definition, but it will be surrounded by a `let`–form which contains the following elements as variables:

1. The names of signals bound to the appropriate index number in the sequence of the signal specifications. (Note that both the name of the first input signal and the first output signal will be bound to 0.)

2. The names of parameters and control parameters bound to the current values of the parameters and control parameters. (Note that in the latter case the value of the control parameter is supplied rather instead of the pointer to the value.)

With the help of this 'let'-form one is able to refer to the signals in the the various MGK functions, and to parameter values in any expression. (See the example.)

4.5.4 Declaration of Compound Modules

A compound can be defined with following form:

```
(defcompound
   Name
   Io_interface Param_interface
   Control_interface
   Environment_interface Task_interface
   Icon Picture
   Keyworded_list )
```

The `Io_interface`, `Param_interface`, `Control_interface`, `Environment_interface`, and `Task_interface` are of the same form as in the case of

the primitives. The only exception is that in the case of environments and tasks there can be many symbols in the lists, and they do have meaning. They will be determined when the structure is generated, as it will be discussed later.

The elements of the `Keyworded_list` are processed when the structure is built and this processing takes place in a sequential manner. This list consists of elements which are lists of the form :

```
(keyword argument_list)
```

The possible forms of the keyword - argument list are the following:

```
(signals Signal_spec_list)
```

The `Signal_spec_list` is a list of signal specifications having the same syntax as in the case of the `Io_interface` the `Input_signal_list` and the `Output_signal_list`. These declarations are the internal signals of this module, which are used only by its internal components. When the actual module description is interpreted, the corresponding datanodes are generated and they will be 'passed' to the sub-modules.

```
(params Parameter_spec*)
```

The `Parameter_spec` is either a symbol (which denotes a parameter initialized to NIL), or a list of two components, where the first symbol denotes the parameter name, the second (being an expression) denotes the initial value for that parameter. These parameters are local static parameters, and are created when the module is interpreted. They can be set, modified and passed to the sub-modules as static parameters.

```
(shared Parameter_spec*)
```

This is the form for locally declared dynamic parameters. The same manipluations can be done with them as with the static parameters.

The `Variable_spec*` denotes variable specifications of the form:

```
Variable_name
```

or

```
(Variable_name Initial_value)
```

The purpose of this declaration is the creation of local variables for the interpretation process. These variables are created when the structure is interpreted. They will have an initial value `NIL` if otherwise it is not specified. References to these variables are legal only in the body of the compound structure (e.g. in the elements of the keyworded list). They preserve their values in run-time also, and can be accessed with the help of the monitor (see it later).

`(compute S-expressions)`

This is an implicit `progn` where the S-expressions are evaluated at the time of the construction. The S-expressions might refer to any element of the interfaces, to the declared local signals, parameters, variables or sub-structures. The expressions are evaluated in the order they were defined.

`(struct Component_specification*)`

This is the specification which describes the internal structure of the compound structure. A `Component_specification` is a list of lists in one of the following two forms:

- Simple form:

```
(Component_name
  (Type_name Io_list
   Parameter_list Control_list
   Env_list Task_list))
```

- Conditional form:

```
(Component_name (
   (Cond-1 (Type_name-1 Io_list
               Parameter_list
               Control_list
               Env_list Task_list))
   ....
   (Cond-n (Type_name-n Io_list
               Parameter_list
               Control_list
               Env_list Task_list))
 )
```

The first form is for the declaration of fixed structures while the second one provides a way to make design decisions at the time of generation. The `Type_name` determines the type of the component. It must be the name of a primitive or a compound. The `Component_name` will be bound to the newly created substructure in the scope of the body of the compound. It is an entity inside of the monitor which contains the structure in the Multigraph sense together with all of the information about it. This structure can be manipulated by the monitor later. The `Io_list` is a list of signals of the form similar to that of the `Io_interface`, except that here only names are allowed and no type specifications. The names must be names of signals either coming from the `Io_interface` or declared locally. The `Parameter_list`, `Control_list`, `Env_list`, and `Task_list` contain symbols which

denote the parameters, control parameters, environments and tasks for the component. Parameters should either be local or coming through the interface of the compound, the environment and task names should come from the interface. The elements of the lists are passed by position and the lists are checked with the declarations of the substructures.

The simple form generates always the same structure as a component. The conditional form takes the conditions, evaluates them, and selects the structure for which the evaluation succeeds. If there is no succesful condition there will be no structure generated. The conditions might contain any expression containing references to parameters, variables, etc.

Thus an HDL structure can be built from the declarations as follows:

- The top–level compound declaration and the values of its parameters should be specified.
- The HDL interpreter takes that declaration and instantiates it with the actual parameter values.
- Then the interpreter recursively calls itself to build the component structures. The recursion stops when a primitive is encountered, in this case an actornode is created.
- The HDL interpreter returns a pointer to the root of the tree–structure built.

This building process can be initiated programatically or through a user interface.

4.5.5 HDL Programmatic Interface

This interface contains a set of Lisp functions which are exported from the HDL package. The HDL declarations should be loaded into the Lisp environment just like any other Lisp program, the HDL interpreter builds its internal databases at load–time.

The functions of the programmatic interface are as follows:

```
(get-hdl-def Name)
```

This functions returns the HDL declaration in the form of an S–expression, which is associated with `Name`. It signals an error if there is no declaration.

```
(create-hdl-struct Name
   Inputnodes Outputnodes
   Parameters Controls
   Environments Tasks)
```

This function builds the HDL tree starting from the declaration identified by `Name`. The lists `Inputnodes` and `Outputnodes` contain datanodes which serve as input and output connection points to the structure. The `Parameters` and `Controls` specify the list of values for the static and dynamic parameters of the structure, while the list `Environments`

and `Tasks` should contain MGK environments and tasks. The function returns an HDL object which contains the instance of the declaration and all its components. This object is used in the remaining functions. After generation all the actor– and datanodes of structure are inactive and blocked. Before starting the structure for the first time, it is advisable to issue a `(init-kernel)` command.

```
(part-of-hdl HDL-object . Names)
```

This function returns a component object of an HDL object, it should be identified by the path of names which leads to it in the hierarchy. (It is similar to the directory access structure of Unix.)

```
(read-hdl-signal HDL-object Name)
```

This function evaluates the `(dnode-value)` function (of MGK) in the specified object as environment, and the named signal as datanode.

```
(write-hdl-signal HDL-object Name Value)
```

This the corresponding functions to do a `(dnode-write)`

```
(get-hdl-signal HDL-object Name)
```

This function is similar to `(read-hdl-signal)`, but it sends requests to the datanode and runs the network until a value is produced.

```
(read-hdl-param HDL-object Name)
```

Retrieves the value of a dynamic parameter in the specified structure, which has the corresponding name.

```
(write-hdl-param HDL-object Name Value)
```

Modifies the value of the dynamic parameter.

```
(activate-all HDL-object)
```

This function can be used to activate all the actornodes in the HDL structure.

```
(deactivate-all HDL-object)
```

The "inverse" of the previous function: it deactivates all the actornodes.

```
(enable-all HDL-object)
```

To enable all the datanodes in the HDL structure one has to use this function.

```
(disable-all HDL-object)
```

This function is for disabling all the datanodes in the network.

```
(eval-in-context HDL-object Expression)
```

The expression is evaluated in the context of the current HDL object. Note that the context binds the symbols appearing in the declaration.

```
(hdl-init)
```

Initializes the HDL system. Must be called before any other function.

4.5.6 Monitor

HDL is equipped with an interactive monitor program, which makes it possible for the user to control the creation and execution of the structures. The monitor has several modes in which the user can communicate with the system through menus and short commands.

The monitor can be started by calling the Lisp function:

```
(hdl-mon)
```

The function displays a menu with the following choices:

```
0)  Exit
1)  Cleanup
2)  Load defs
3)  List names
4)  List def
5)  Generate
6)  Set env prior
7)  Show hierarchy
8)  Show current names
9)  List current definition
11) Change focus
12) Control structure
13) Manipulate structure
14) Eval in current structure
```

The user should select a command by typing the corresponding number.

The commands are as follows:

- The "Exit" command returns to the Lisp top-level loop.

- The "Cleanup" clears the internal tables of the monitor, initializes MGK, and removes all the definitions. It can be used for example if somebody wants to load in a new definition file which contains definitions for previously defined things. This command MUST be used before using any other facilities of the monitor.

- The commands "Load defs", "List names", "List def" are for loading Lisp files, listing the names of primitives and compounds known to the system, and printing out a declaration, respectively. File names should be specified as Lisp strings, while names for declarations as Lisp symbols. [1]

- The "Generation" command lets the user build an HDL structure. The necessary things to be specified are: (1) the name of the top–level declaration, which has to be a compound, (2) the initial values for the parameters, and (3) the names of tasks and environments. [2] Note that before generating anything, one has to load all the declarations which are needed.

- The "Set env prior" command lets the user change the priority of an environment. The users is asked to enter a line of the form:

  ```
  Environmentname Integer
  ```

- The "Show hierarchy" command displays the tree of HDL objects. The tabulation shows how the structures are nested.

  The monitor uses the concept of focus: it means the HDL object, what is currently accessible to the monitor commands. Initially the focus points to the root point of the HDL object tree, but the user can change this. The focus also provides a lexical context, in which things are to be looked up. Because things can be called by the same name on different levels of the hierarchy, the context provided by the focus is essential.

- The "Show current names" command lists the symbols meaningful in the context of the focus, while "List current definition" prints the S–expression of the declaration which describes the focus.

- To change the focus one can walk up and down the hierarchy, using the "Change focus" command. The command shows the hierarchy, with an arrow where the focus is. Then the user is asked to enter a command, which can be a number or a letter. If it is a number, say "n", then it means the n–th subcomponent of the focus. (The numbering starts from zero.) The following letters are interpreted:

  – ? : Help

[1] Note that in the latter case sometimes the vertical bar should surround the symbol.

[2] HDL uses a task representation package, and for the top–level of the builder the task and environment names have to be specified.

- ^ : One level up
- r : Redraw hierarchy
- q : Quit this command

This way one can access arbitrary components in the hierarchy by navigating in the tree of HDL objects.

- Before running an experiment certain MGK commands should be executed on the actor and datanodes of the system. This can be done using the "Control structure", which lets the user activate/deactivate (enable/disable) the actor (data) nodes. These actions take place in the object identified by the focus, i.e. in the subtree, where the root is the focused object. The commands are as follows:
  - "a": Activate actornodes
  - "e": Enable datanodes
  - "d": Deactivate actornodes
  - "b": Disable (block) datanodes
  - "q": Quit this command
- When the components of the network are enabled and activated the system is ready for execution and run-time manipulation. The command "Manipulate structure" starts a command line interpreter, through which the user can handle the structure. The following commands are known:
  - `r Signalname`
    Read the value of the signal object (datanode) and print it.
  - `o Signalname`
    Obtain (i.e. force out) the value of the signal object and print it.
  - `w Signalname Value`
    Write a new value into the signal object. `Value` is evaluated by the Lisp interpreter.
  - `g Parametername`
    Get the value of the named control parameter and print it.
  - `s Parametername Value`
    Set the value of the control parameter, `Value` is evaluated.
  - `q`
    Quit this command.

Note that all these commands are evaluated in the context of the focus, i.e. signal names should denote signal objects in the current context.

- The "Eval in current structure" starts up a read–eval–print loop, which evaluates the expression typed in the current context. If the expression is an empty line, it returns to the menu. If an error is detected during evaluation, the main top–level comes back, from which one can go back to the monitor by restarting it using the "(hdl-mon)" command.

4.5.7 An example

```
;;;
;;; Quite complex example for hdl using Lisp
;;;

(defprimitive Random :Ifany
  (Start -> Out) (Length) ()
  (Env) (Task) Icon Picture
  (let ((res (make-array (list Length))))
   (Receive Start)
   (dotimes (i Length)
      (setf (svref res i) (random 1.0)))
   (propagate Out res)))

(defprimitive Ramp :Ifany
  (Start -> Out) (Length) (Offset Slope)
  (Env) (Task) Icon Picture
  (let ((res (make-array (list Length)))
        (tmp Offset))
   (Receive Start)
   (dotimes (i Length)
      (setf (svref res i) tmp)
      (setf tmp (+ tmp Slope)))
   (propagate Out res)))

(defprimitive Mult :Ifall
  (In1 In2 -> Out) (Length) ()
  (Env) (Task) Icon Picture
  (let ((in1 (Receive In1))
        (in2 (Receive In2))
   (res (make-array (list Length))))
   (dotimes (i Length)
      (setf (svref res i)
            (* (svref in1 i) (svref in2 i))))
```

```
   (propagate Out res)))

(defprimitive Fft :Ifall
  (Realin Imagin -> Realout Imagout)
  (Length) ()
  (Env) (Task) Icon Picture
  ;;; Code for FFT has been omitted
)

(defcompound Generator
  (Start -> Result) (Length) (Offset Slope)
  (Env) (Task) Icon Picture
  (signals Rampsignal Rndsignal)
  (struct
    (Rmp-gen
      (Ramp (Start -> Rampsignal)
            (Length) (Offset Slope)
            (Env) (Task)))
    (Rnd-gen
      (Random (Start -> Rndsignal)
              (Length) ()
              (Env) (Task)))
    (Mul
      (Mult (Rampsignal Rndsignal -> Result)
            (Length) () (Env) (Task)))))

(defprimitive Printer :Ifall (Real Imag)
  (Length) ()
  (Env) (Task) Icon Picture
  (let ((rbuf (Receive Real))
        (ibuf (Receive Imag)))
   (format t " Printer ~%")
   (dotimes (i Length)
     (format t "R-~S : ~S~%" i (svref rbuf i)))
   (terpri)
   (dotimes (i Length)
     (format t "I-~S : ~S~%" i (svref ibuf i)))
   (terpri)))

(defcompound Test (Start ->)
  (Length) (Offset Slope)
  (Env) (Task) Icon Picture
```

```
(signals Rsignal Isignal Rfreq Ifreq)
(struct
  (Real-gen
    (Generator
      (Start -> Rsignal)
      (Length) (Offset Slope) (Env) (Task)))
  (Imag-gen
    (Generator
      (Start -> Isignal)
      (Length) (Offset Slope) (Env) (Task)))
  (Fft-op
    (Fft (Rsignal Isignal -> Rfreq Ifreq)
         (Length) () (Env) (Task)))
  (Print-it
    (Printer (Rfreq Ifreq)
         (Length) () (Env) (Task)))))
```

4.6 Interfacing of HDL to *Agent*

The *Agent* package has a command interface which can be used by another testbed component like HDL. The general technique of the interfacing is the use of *pipes*, which are supported by the workstation's operating system. The knowledge-based component starts up the graphical simulation package and establishes communication with it using the pipes. The pipes look like files for the knowledge-based component, where it can write into and read data back from.

The following two low-level Lisp functions are provided:

```
(popen "programname")
```

Starts up the program called "programame" (typically: \verb"agent"+), and builds a pipe connection to it. The function returns a two-way stream which can be used for input/output in the Lisp program, whatever is written into the stream will be sent to the started program's standard input, and whatever is sent to the standard output of the program can be read from the stream.

```
(pclose pipe)
```

Closes the pipe connection to the running program. Note that the user must force the (agent) program to exit before closing the connection (by, e.g. sending an `exit` command).

These two low-level functions supply the interface for starting the graphical simulation and interfacing it to the knowledge-based layer. Using standard IO facilities arbitrarily complex communication schemes can be built, for example: a HDL script can send control commands to the graphical simulation and receive back position data, etc. In the next chapter we show an example system which was built using a communication scheme.

Chapter 5

Case Studies

The modeling methodologies and tools described in the previous chapters provide a usable working environment for testing automation concepts regarding space applications. This chapter describes some of this work. First a structural, geometric model of the Space Station is presented, which was prepared using the graphical modeling techniques of the previous chapters. Next, another modeling efforts are described which combine this graphical model with knowledge-based techniques to simulate various operational aspects of the Station. Part of this work was the modeling of the Space Station Environment Control and Life Support System (ECLSS) which was performed using the symbolic modeling techniques introduced in the previous chapter.

5.1 Space Station Modeling Using ROBOSIM

In the last three years, ROBOSIM has been applied in numerous occasions to develop and study real-time models of industrial manipulators. It's use, however, was not limited to robotics only. Recently, ROBOSIM was put to use to support a sequenced build-up of the space station model. The porting of ROBOSIM to a real-time graphics workstation, the HP 350SRX, with it's 3D graphics capabilities, knobs and menus served as a more interactive and user-friendly tool which allowed for superior illustration and detailed examination of different parts of the space station model.

In designing the space station, just like in designing a robot, the selection of the robot's kinematic design is usually considered first. The number of robot joints, type of joints(rotational, sliding or fixed) and the physical configuration are all important factors of the robot's kinematic design.

After careful analysis of NASA's latest configuration of the space station model (SS, for short) and knowing ROBOSIM's capability of handling multiple number of manipulators within the same working plane, a modular approach was chosen to construct the SS model.

The SS model was broken in to several independent, serially linked manipulator models, all assigned the same reference frame. Each manipulator consisted of separate parts, where each part was built as a compound object made of primitives, such as boxes,spheres,

cylinders and user-defined shapes. These parts were then assigned the correct kinematic parameters and mass properties and finally assembeled together using ROBOSIM.

The modular approach was a necessary approach as well as a practical one. It was necessary, because it helped overcome the problem of serial-linkage, usually associated with robotic simulation packages, where a movement in one link will cause a movement in the next.

Breaking down the model into separate independent manipulators, helped overcome this obstacle. For example, each set of the solar panel assemblies could now be adjusted and controlled independently of the other set. The modular approach is also practical because it allowed for complicated models to be created in smaller parts and assembled as the designer required. Changes could then be made to any component of the model without affecting other parts. New parts or manipulators could also be added just as easy without affecting any of the existing models.

The SS model was broken into five independent, serially linked manipulators, with each manipulator representing a desired set of rotations and/or translations. These models were represented as follows:

- **Two solar panel structures** (Fig 5.1 and Fig 5.2), each of which is treated as a separate manipulator attached to the side of the main truss assembly. Since both solar panel structures were physically identical, only one structure had to be constructed. The other was simply replicated, but assigned different kinematic properties. This feature of ROBOSIM helped save time and effort, since structures can be saved in a file for later usage.

- **Two identical sets of solar panels, heat radiator and truss assemblies**, each treated as an independent manipulator. These assemblies attach to both ends of the middle truss assembly. Each assembly has two rotational movements. One for the solar panels, to position them in a direction facing the sun, the other for the whole assembly structure to be able to position the heat radiators away from the sun.

- **Mobile servicing robot**, with five rotational joints, sliding on a set of rails to be attached to middle truss assembly (Fig 5.3). The robot was modeled as a six degree of freedom manipulator, with the sliding rails serving as a translational joint. The robot is used to perform routine tasks, e.g., inspection and maintenance.

- Finally, **the middle truss assembly** was built. It included crew living modules, antennas and truss assemblies all attached together to create the main body, to which all other sub-models attach (Fig 5.4). A common frame, to which all other manipulator models refer, was assigned at the middle of this truss assembly.

- For easier debugging, this final structure was broken into three parts: **Crew-living modules, antennas and trusses.** A set of two non-shaded, user-defined cubes were built and propagated, using temporary storage registers, to construct the truss assembly. This illustrates ROBOSIM's ability to create user-defined shaded and non-shaded objects as parts of the same model.

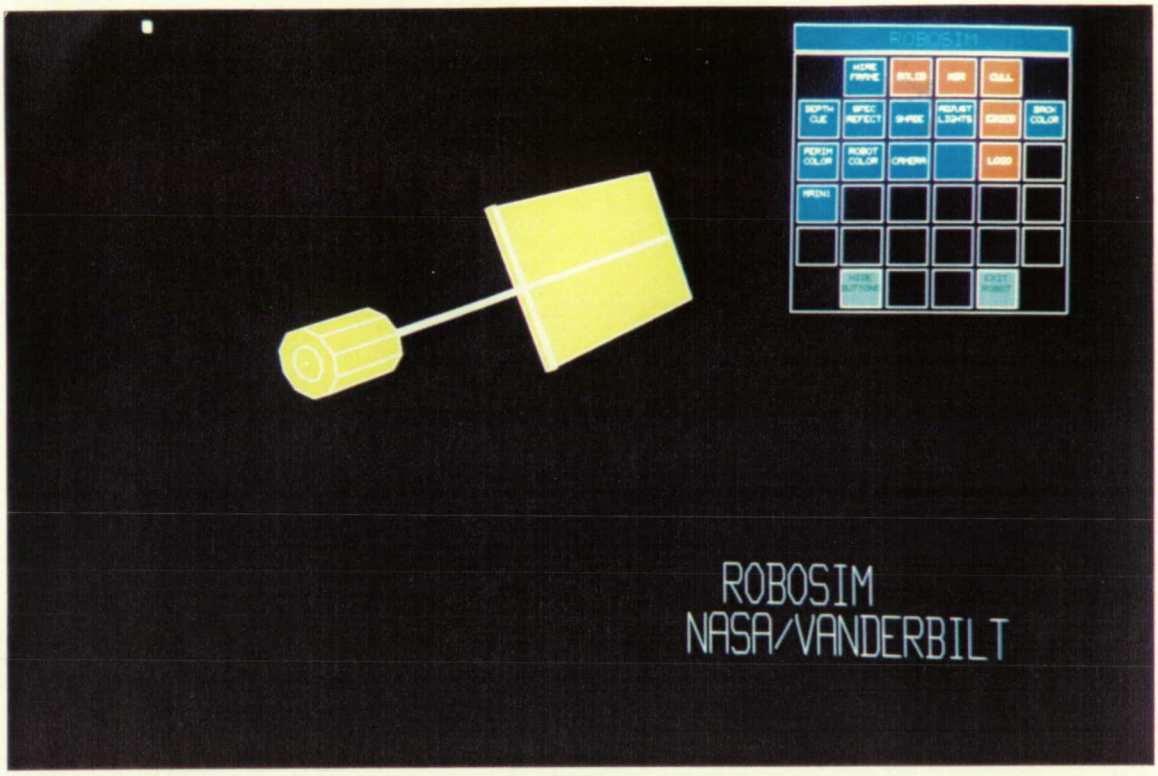

Figure 5.1: Solar panels - I.

With the links of all five models being defined and a common reference frame assigned, the graphics display program was used to assemble the different parts, in a pre-assigned configuration, to generate the desired SS model (Fig 5.5) The menu box, top right of the screen, provides various options with which the user can interactively view and control separate parts of the model.

Separate routines could also be linked to the Graphics display program to assign joint limitations and/or set motion along any parametrically defined functions. Two sets of predefined motion for the main solar panel assemblies is shown in Fig 5.6.

5.2 Operational Modeling of the Space Station

Space Station automation requires the analysis of the complex material, energy and information transfer processes from many different aspects. The structural model introduced previously is just a representation of one of these aspects, but to cover the full range of

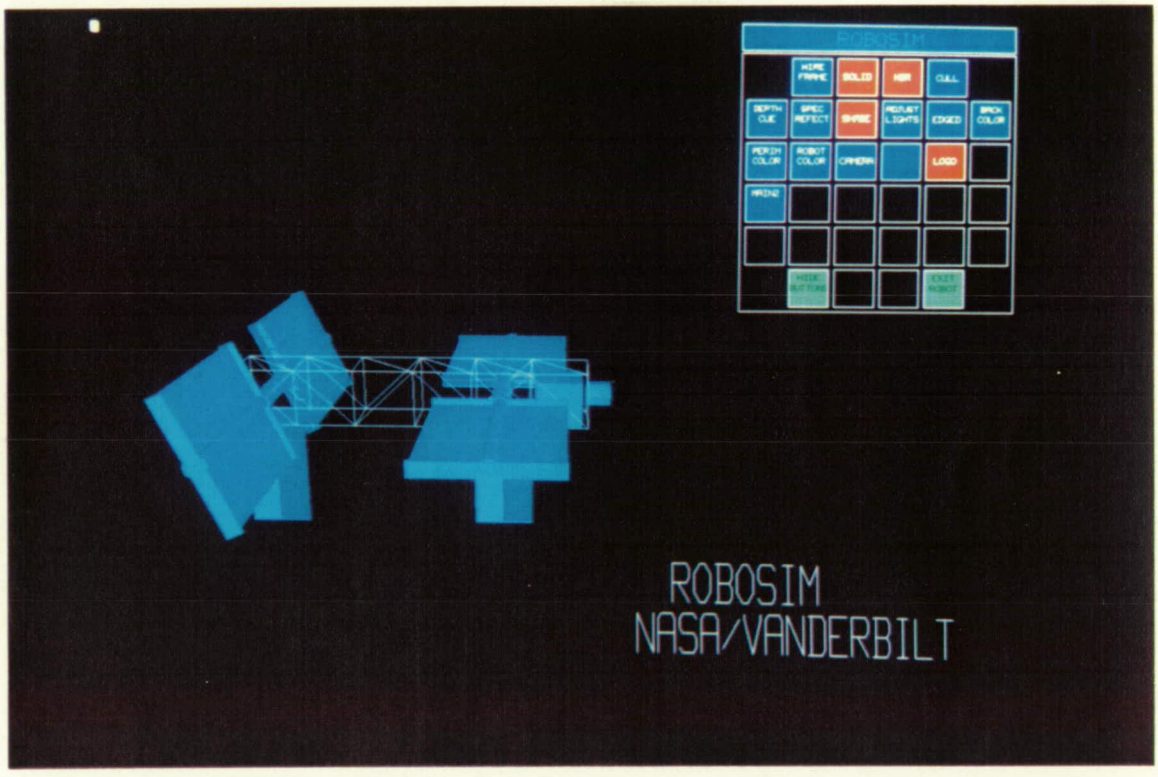

Figure 5.2: Solar panels - II.

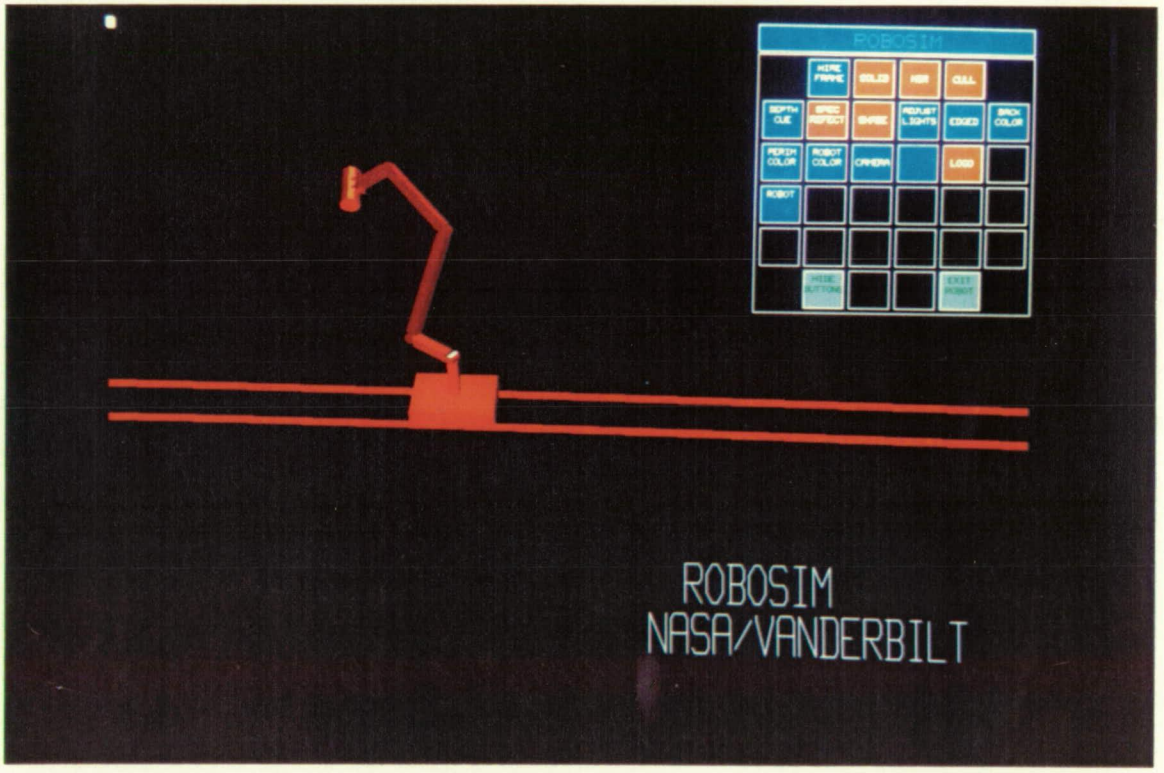

Figure 5.3: Servicing Robot

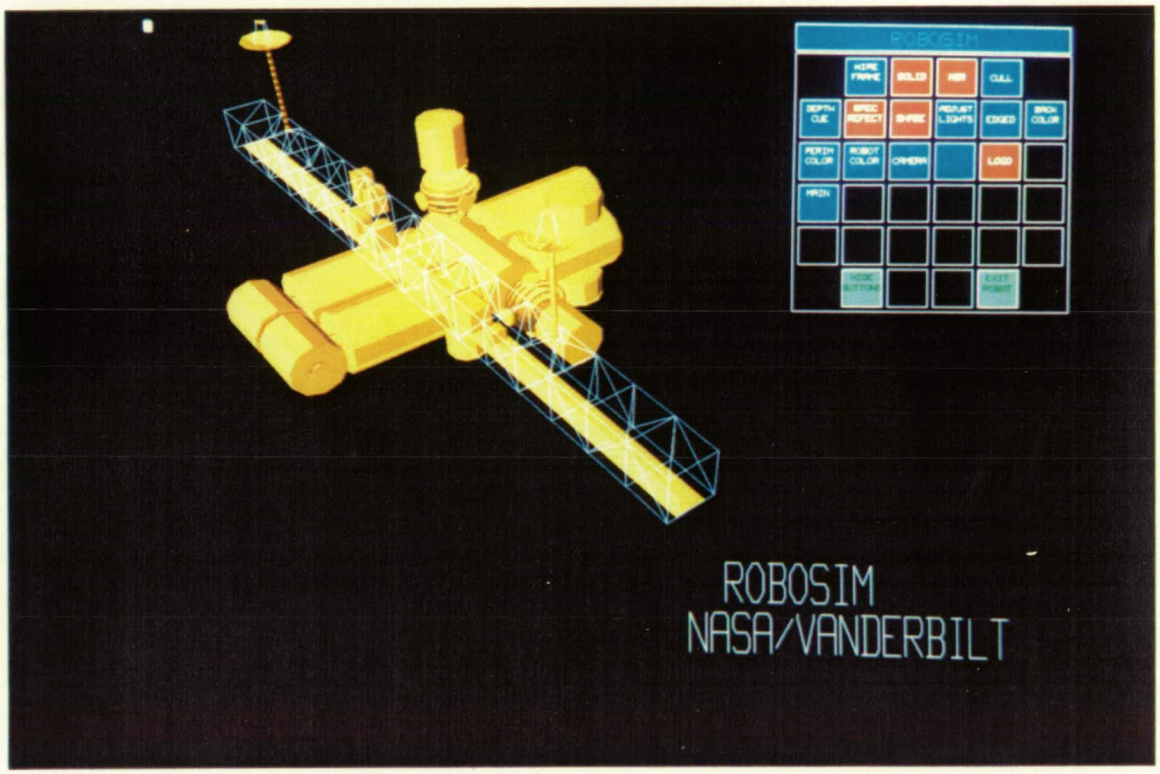

Figure 5.4: Middle truss assembly

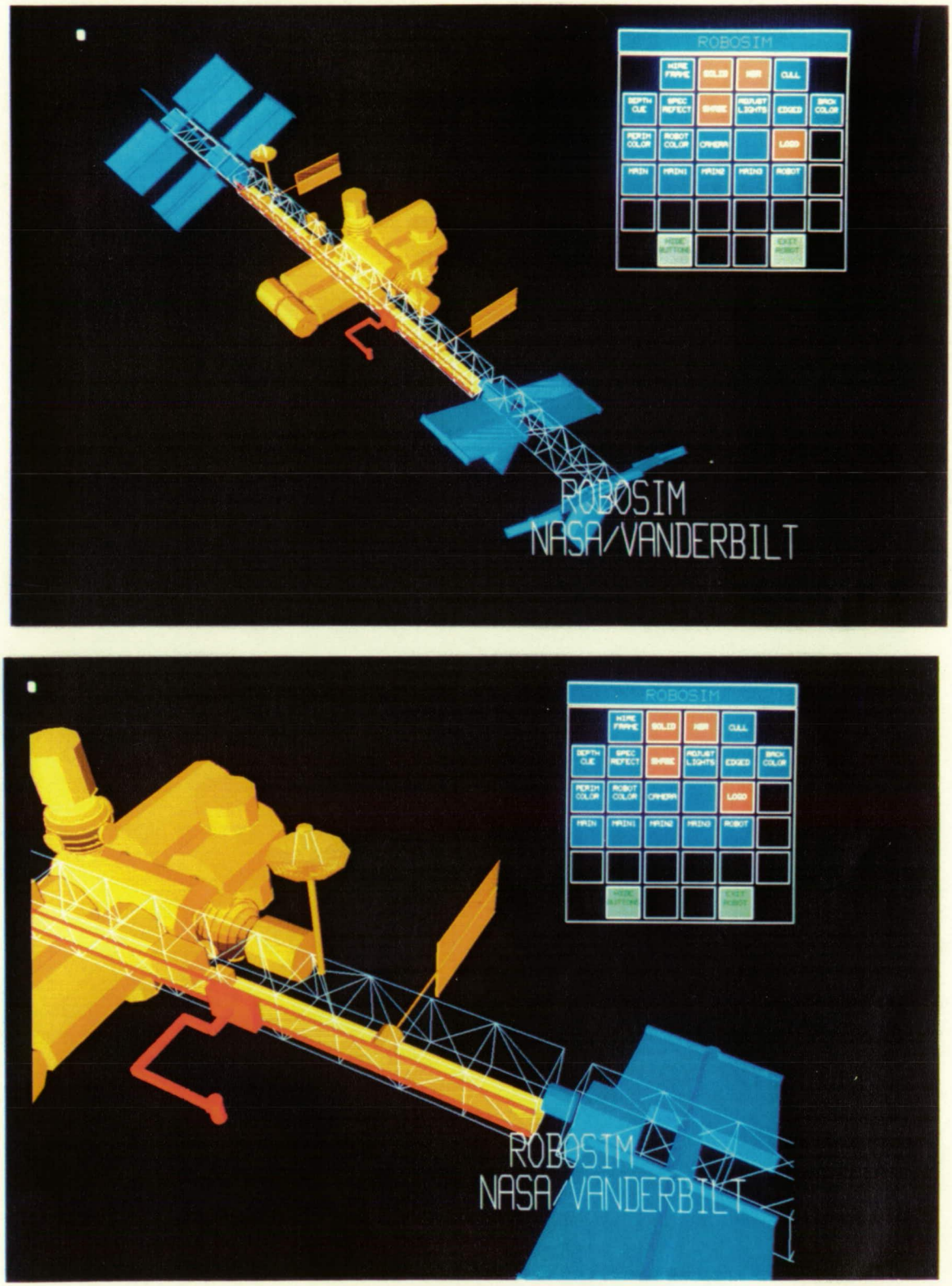

Figure 5.5: Space Station Model

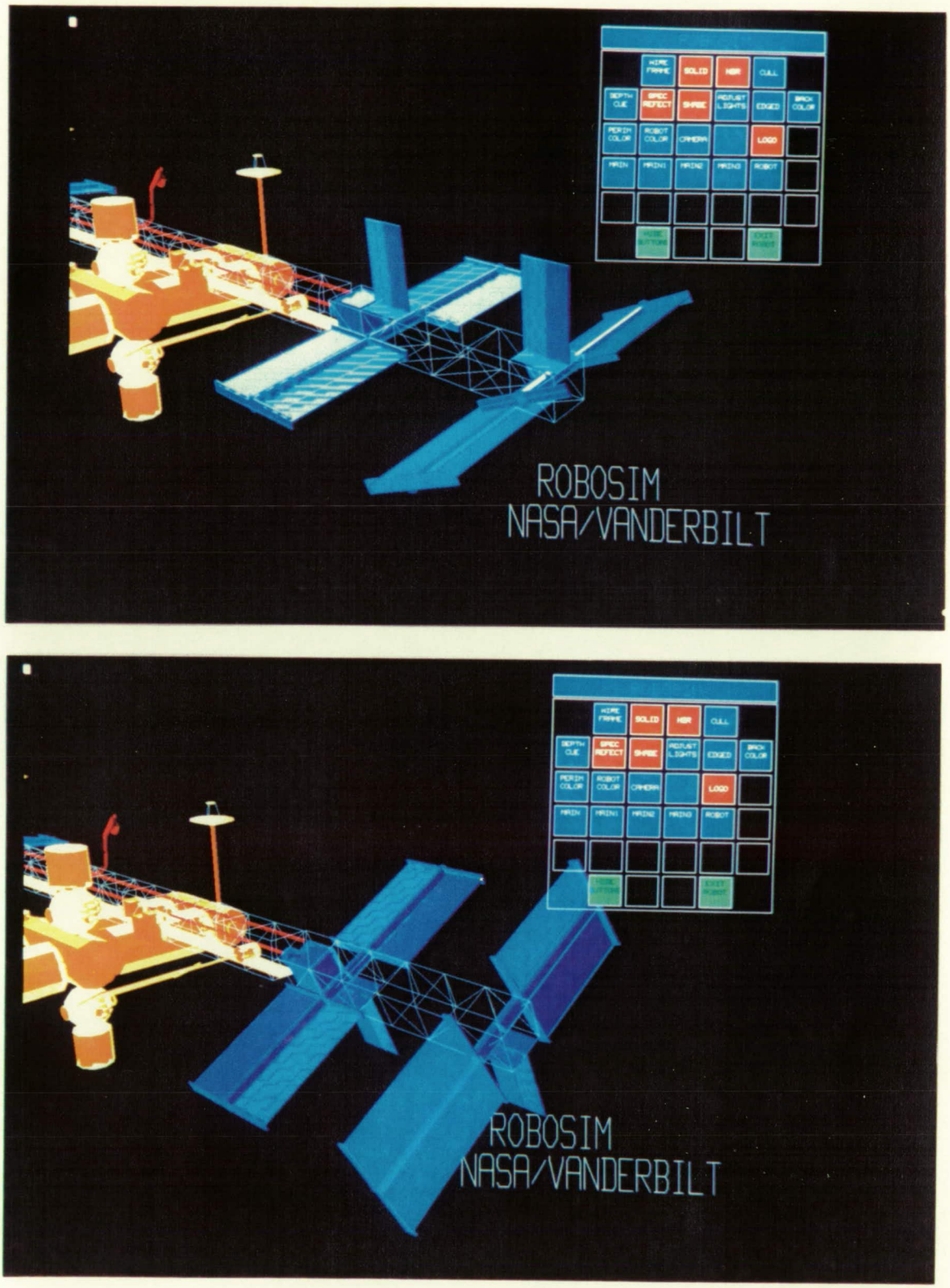

Figure 5.6: Solar Panel Motion

possible operations, it has to be combined with other models representing the different aspects and using different modeling techniques. The integrated modeling environment which was the subjects of the previous parts of this report offers a unique opportunity to do this. Below we list a couple of the problems which are well suited for this approach.

Attitude Control System and its Dependencies: The Space Station is a large structure, which due to the different disturbing effects (solar wind, etc..) requires a constant control of its attitude. This is done by a triple gimbaled gyrator system (according to the plans). The structural model of the station together with the (already existing and newly developed) elements of the Simulation Library could provide a toolkit to test the orbital mechanics and the attitude control problems related to the station.

But this is just one of the aspects of the attitude control problem! The Space Station is a relatively small closed system, so everything influences everything. Normally the triple gimbaled gyroscopic attitude control system is sufficient to control the orientation of the Station. But during the course of the operation, the rotational angles of gyroscopic wheels might reach a position where they align with each other - which means that the system is not capable of control any more. In such cases the gyroscopes must be 'recharged' i.e. their angles of rotation made (approximately) perpendicular again. This of course will offset the orientation of the Station which then must be corrected using thrusters. It is expected that this operation will have to be performed at about every tenth orbit. There are several constraints which influence this:

- This 'recharging' operation might disturb some ongoing low-gravity experiments (because it introduces relatively high accelerations), so these have to be considered when scheduling it.

- There are other orbital maneuvers which affect the attitude control (docking or launch of objects). A higher-level controller which schedules the recharging activities of the attitude control system must know about these events too.

- While the Station is on the 'sunny' side of the Earth, the photelectric cells should be operating at the possible highest capacity. If the sudden changes in the station's orientation can not be followed by the control system of the panels then the energy production might suffer. On the other hand the operation of the solar panel's alignment mechanism itself influences the attitude control system.

The Electrical Energy Production and Distribution System itself is an interesting area of study, due to the limited energy supply and the interactions between the different consumers. Some of the problems in this area are:

- A control system must be developed which utilizes the periods while the Sun is visible most efficiently by aligning the solar panels as close to perpendicular to the Sun as possible. We have already begun developing a model for such a control system for this purpose utilizing the structural model of the Station and some of the higher-level symbolic tools introduced previously.

- A higher-level controller of this subsystem must predict the future energy production (interactions with attitude control and other orbital operations!), and based on the reserve energy and projected production must schedule the operation of the different consunmers. This seems to be a task to be solved using knowledge-based techniques, possibly by using the modeling techniques of Chapter 4 to simulate the different consumers.

- There are vital subsystems on the Station whose energy demands must be satisfied. An example of these is the Environment Control and Life Support System (ECLSS). Beside being a very important energy consumer, ECLSS is also a big energy consumer. If it is predicted that ECLSS's energy demands can not be met, the whole operation of the Station may have to be rescheduled. Actually ECLSS itself is a set of interrelated subprocesses, some of which are not as important as the others. For example in the case of an energy shortage the air control subsystems for the experiment modules might operate at a reduced capacity, while it is not true for the crew modules. Such a decision will result in having to stop some of the ongoing experiments. But this again is just one of the possible interrelations.

A common characteristics of the above examples is that modeling them requires considering many aspects of their operation. Some of these aspects can be expressed in quantitative terms, while others only in qualitative ones. This fact is the best justification for an integrated automation simulation and modeling testbed, containing (1) geometric, graphical modeling tools for spatial modeling of the different systems of the Station (e.g. ROBOSIM), (2) a model-based programming environment for creating deep, structural, knowledge-based models for adaptive control and failure analysis (e.g. MULTIGRAPH).

5.3 Study of the Space Station ECLSS

One of the most important systems of the Space Station is the Environment Control and Life Support System (ECLSS). This is a vastly complicated system with many interacting subsystems. Design of low-level control systems for these subsystems is based on modeling the process dynamics. Development of a ***diagnostic system*** requires the elaboration of sophisticated fault models, and the construction of the operator interface is closely related to various qualitative models of the subsystems. The analysts can develop these models of different levels of abstractions, and can apply them for a particular purpose.

Due to the difficulty of these problems, the support of modeling is of paramount importance in a simulation testbed for automation. The purpose of this case study was twofold: (1) to demonstrate the use of a multiple-aspect modeling technique in analyzing the diagnosability of the ECLSS, and (2) to demonstrate the integration of structural, geometrical modeling with the knowledge-based components and diagnostics system. This latter demonstration has produced an integrated system fault diagnosis and repair actions are tightly coupled. The study was being conducted in close cooperation with the Boeing Aerospace Company, Huntsville, Al.

5.3.1 Objectives of the ECLSS study

ECLSS is a large system comprising complex material, energy and information transfer processes. The primary tool for the design and operation of the system is extensive modeling. The models help to understand the ECLSS in the design phase, and they are the key components of the monitoring, diagnostics and control system in operation time.

From a methodological point of view, we consider the ECLSS design process as an incremental model building activity, in which various system components are defined in terms of specific models. The design is successful if the individual models are correct, and if the various models are consistent with each other. If the progress in the design process is represented in the form of a set of formal models (quantitative and qualitative), the intermediate results can be tested and validated by using the following techniques:

- The consistency of the models of different levels of abstraction can be tested by using mapping rules among the modeling aspects.
- The models can be used for the generation of quantitative/qualitative simulations of the system, in order to test its expected behavior from a selected aspect.
- The performance of specific subsystems (e.g. diagnostics, or control) can be tested in a simulated environment.

AI provides a rich selection of modeling techniques that can support this process. Knowledge representation techniques can be developed to describe qualitative and quantitative features of systems. These representations can be used to test the correctness of the individual models, to check the consistency among the related modeling aspects, and to analyse different features of the system designed.

The objective of the study was to test the integration of real-time *fault diagnostics* with a *fault recovery facility* for the ECLSS Potable Water unit. The specific objectives were the following:

- *Multiple-aspect modeling of ECLSS Potable Water Unit.* The models define the energy, material and information processes in the system in a hierarchically organized way. These models include the **Hierarchical Process Models** (HPM) which serve as the **dominant modeling aspect** for the study. HPM provides the context for other, **dependent modeling aspects.** The structure of the physical processes in ECLSS are modeled by using the graphic/declarative modeling techniques of MPE.
- *Hierarchical Fault Models (HFM) of ECLSS Potable Water Unit.* The fault models specify **fault modes** and **fault propagation paths.** The structure of the fault models corresponds to the structure of the process models, since faults can propagate only through physical interactions that are expressed in the process models. The **multiple aspect** modeling methodology of MPE ensures the consistency between the process models and the fault models.

- *Integration of fault recovery mechanisms into the models.* Whenever the fault diagnostics detected a fault and identified the fault source, it should trigger a fault recovery action. The fault recovery mechanism, in this demonstration, is coupled to a robot system, which performs the necessary repair action(s). The robot system can be driven (1) directly via commands incorporated in the models, or (2) indirectly via commands generated by a high-level robot planner. The robot itself can be a (1) physical robot, or a (2) simulated robot. In the demonstration described below we have used direct commands to drive a simulated robot created using the Robosim environment described in the previous chapter. In a system created at Boeing Aerospace an actual physical robot was controlled by using a high-level planner system.

Although, the study is limited to the issues of integration of diagnostics with automated fault recovery, we can easily expand the system later with other modeling aspects, such as modeling the monitoring system, operator interface, control system, etc. By using the automatic program generator services of MPE, the models can be used for generating an executable version of these sub-systems.

5.3.2 Model-based diagnostic system

In this ECLSS study we have used a sophisticated model-based diagnostic system, which applies a hierarchically organized fault propagation model. In this section we summarize the properties of the diagnostic system and discuss the specification of the fault model.

A real-time fault detection and diagnosis capability is absolutely crucial in large-scale space systems. Some of the existing AI-based fault diagnostic techniques like expert systems and qualitative modeling are frequently ill-suited for this purpose. Expert systems are often inadequately structured, difficult to validate and suffer from knowledge acquisition bottlenecks. Qualitative modeling techniques often generate a large number of failure source alternatives, thus hampering the speed of the diagnosis.

A Hierarchical Fault Model of the system to be diagnosed was developed. At each level of hierarchy, there existed fault propagation digraphs denoting causal relations between failure-modes of subsystems. The edges of such a digraph were weighted with fault propagation probabilities and fault propagation time intervals. Efficient and restartable graph algorithms were used for on-line, fast identification of failure source components.

A real-time fault diagnostics system has to function in an environment where new alarms may constantly be generated, due to the propagation of failures. To cope with such a time-changing scenario the diagnostics system must have the following characteristics:

- Signal Processing, Alarm Generation and Failure Source Identification software must be as fast as possible. The first two are usually standard well-defined and analyzed algorithms, and hence, virtually all speed improvements have to be achieved in the failure source identification phase.

- The diagnosed results must be updated as time elapses and new alarm information is received. These results must be accurate but need not have a fine resolution. This implies that in the early stages of diagnosis a large component such as the Potable Water Assembly can be identified as the fault source. The resolution of this fault source is further refined with the passage of time and additional alarm information to a unique component inside the Potable Water Assembly.

- The User-Interface must present the current status of diagnosis in a comprehendible manner, reflecting the level and the granularity of the system under diagnosis, at which the diagnostics system is operating.

The basic philosophy of the graph-based approach is based upon multiple-aspect modeling. The system under consideration is hierarchically decomposed from many aspects in order to yield a set of different models. The **functional decomposition** leads to the Hierarchical Process Model (HPM) and a structural decomposition leads to a Hierarchical Physical Component Model (HPCM). A Hierarchical Fault Model (HFM) is developed in the context of HPM with links to the HPCM.

The technique of hierarchical decomposition is widely used during model building for the following reasons:

- Design, knowledge acquisition, and knowledge-base maintenance of large complex systems becomes structured and easier.

- Running the same graph algorithms on smaller number of nodes many times takes lesser time than running them on the entire set of nodes in a system. For example it takes a longer time to run an $O(n)$ algorithm on a graph with 200,000 nodes than it takes to run the same algorithm 200 times on a graph with 100 nodes.

- It is possible to conduct the search for the failure source on the HFM in a parallel manner, thus enabling speedy diagnosis.

- In most cases a large granularity component assembly can be identified as a failure source at an early stage, and then the search needs to proceed only in that component's part of the model.

A process in the HPM can be thought of as a functional unit carrying out a specific function in the system, by utilizing different physical components. Different processes on the same level may interact with each other through shared physical components. Processes in the HPM can be associated with many different components in the HPCM as shown in Figure5.7. In the context of each process the following are acquired:

- Process Failure-Modes.

- Process Alarms and alarm-generators. The alarm-generators accept sensor inputs and if needed, generate the appropriate alarm.

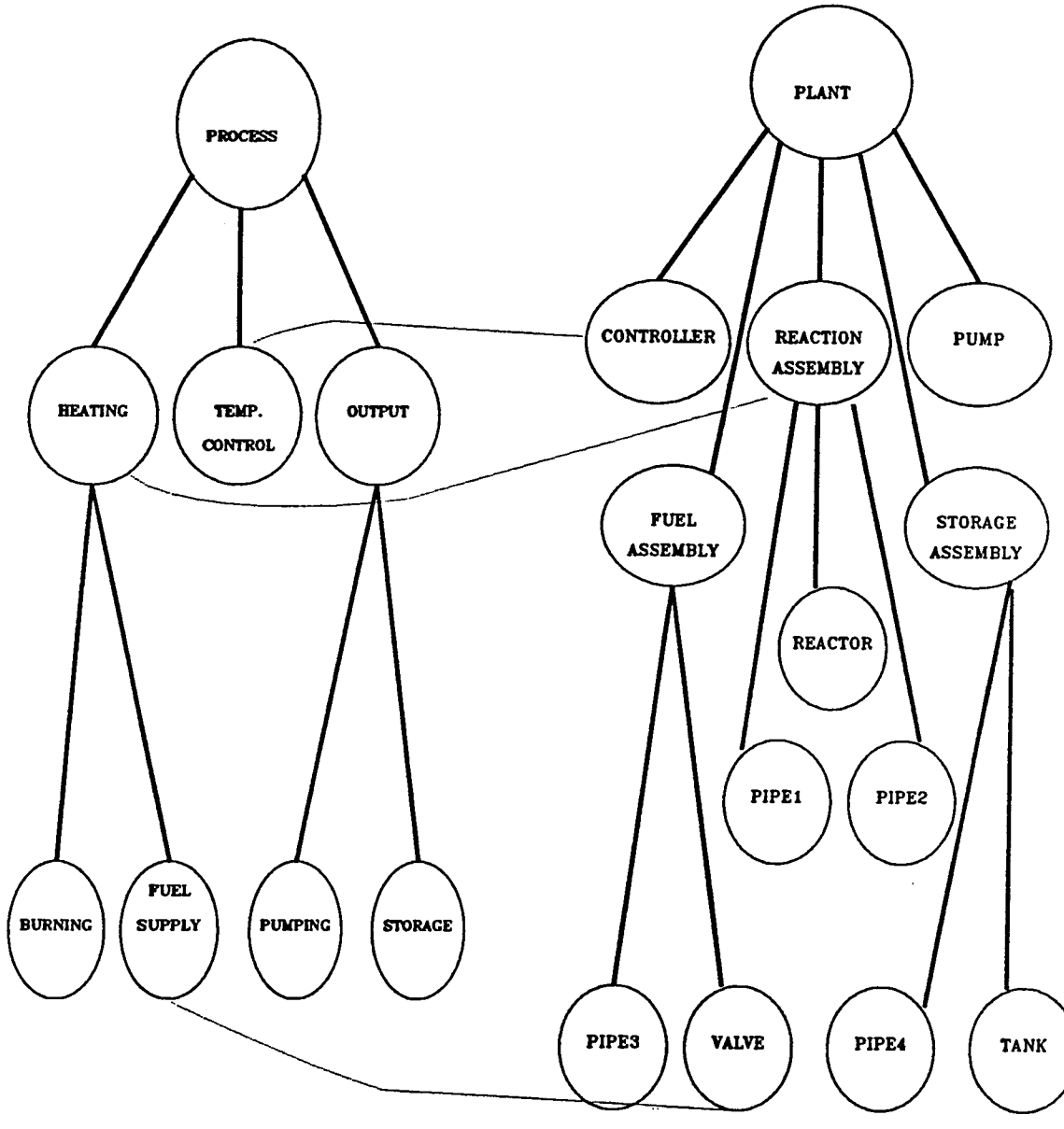

Figure 5.7: A Hierarchical Process Model

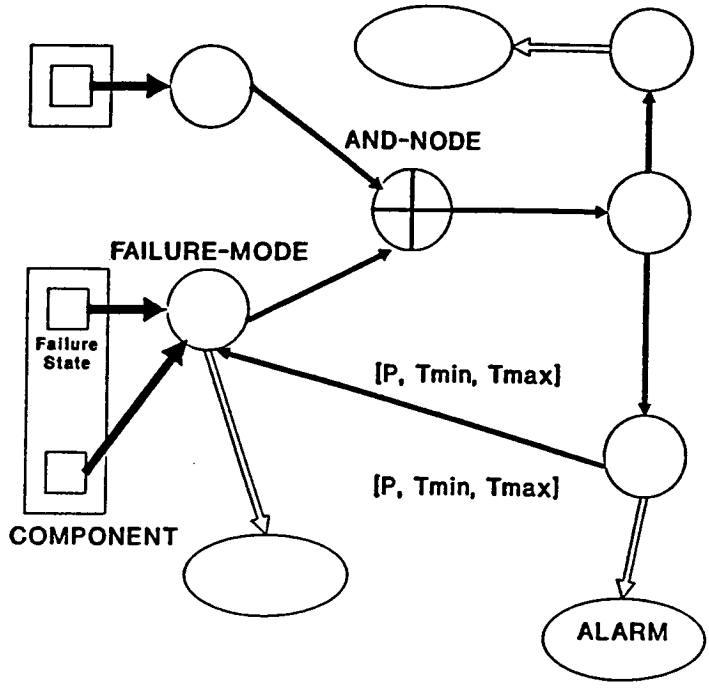

Figure 5.8: Fault Propagation Digraph of a Process

- Alarm Failure-Mode associations.
- Failure-Mode Physical Component associations.

Each process in the the HPM has its fault model, therefore fault models are considered to be dependent aspects to the process models. This model is determined by the failure-modes of the process, and if present, the failure-modes of its subprocesses. All these failure-modes form nodes of a fault propagation digraph, with directed edges between individual failure-modes signifying a fault propagation possibility. Each edge in this graph is weighted with two parameters a fault propagation probability and a fault propagation time interval in terms of a minimum and a maximum. The fault propagation digraph of a process on level i is shown in Figure 5.8. The collection of all such fault propagation digraphs and failure-mode physical components associations results in the HFM. It is possible to extract the basic structure of the fault propagation digraph from the process models, since most faults can only propagate along physical connections.

5.4 ECLSS Study: Diagnostics and Repair

The information for the ECLSS models was acquired from BAC design engineers. The main steps of the study were the following:

- Definition and refinement of the HPM and HFM for the ECLSS.
- Derivation of a real-time alarm pattern simulator from the HFM. The alarm pattern simulator generates alarm sequences from the HFM by using the fault propagation information in the models.
- Determination of the repair actions for Potable Water Unit. Repair actions are associated with physical component failures: i.e. both the component and its failure state together determine the kind of repair action to be taken.

5.4.1 Process and Fault Modeling for the ECLSS

The starting point for this case study was an informal description of the ECLSS Potable Water Processing functionality in terms of layered process component drawings and a fault diagnosis handbook indicating possible faults of the system or of its subcomponents. The main goal was to show a snapshot of how the ECLSS/PWU is represented and how our technology could be used to obtain problem representations, applicable for a variety of tasks including fault diagnosis.

5.4.2 Hierarchical Process Model (HPM) of the ECLSS

The first step is to obtain a functional decomposition of the system. The decomposition, naturally, should not follow the physical layout but rather the functional layout. The stepwise refinement of the ECLSS Potable Water Processing leads to the process hierarchy tree shown in Figure 5.9. Each node represents a certain function in the system. Each function maps the specified input process variables to the specified output process variables. Process variable dependencies in the hierarchy are possible only between parent and son nodes or between sibling nodes. The dependencies are denoted by (directed) edges, called *connections*. Connections typically describe a material flow.

In the following, the process hierarchy is described in more detail. The top level process is the *Processing-of-Water* process. The functional decomposition of the ECLSS/PWU system consists of three subprocesses:

- Pumping
- Filtration
- Distribution
- Testing

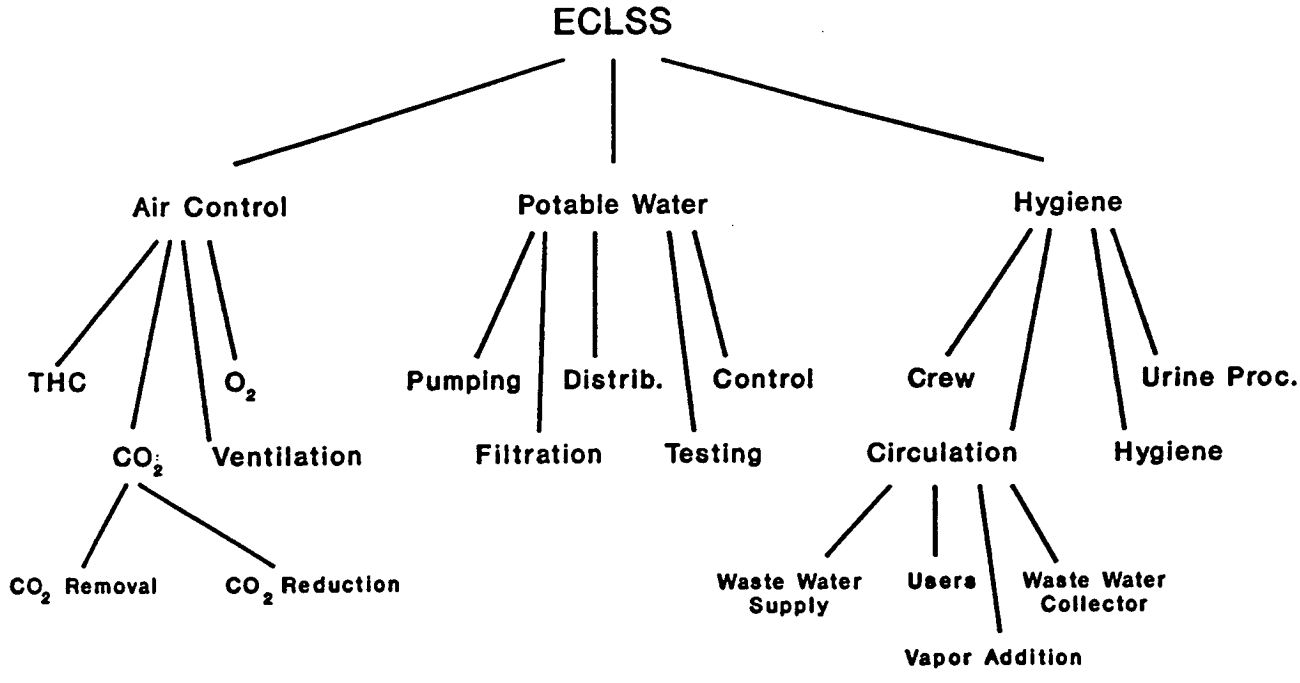

Figure 5.9: Process Hierarchy of the ECLSS

- Control

The physical structure of this process includes two rows of each 4 tanks. At a specific time each tank has a certain unique functionality as collection, storage, or supply. While one of the tank rows is collecting the potable water from the one process the other row is processing the water. Whenever one tank is full the tanks will switch their functionality i.e tested water can be used.

For the functionality of the potable water processing it is not important which tank is used for the filtration or for the water testing. This leads to the fact that on this level of decomposition the physical layout is completely ignored. However the physical layout is important for the entire model and can be modeled as a different view of the system.

5.4.3 Declarative Form of the HPM

The MPE facilitates the modeling process by a graphical editor. The use of graphic editors helps to avoid errors and increases the visibility of the system structure. For further use of the process model a more formalized representation has to be generated. This is done automatically by our system. Figure 5.10 shows the declarative form of the air control process. The declarative form contains all the necessary information to reconstruct the model and the graphical presentation of it. Furthermore the declarative form can be extended to carry more information about different views of the system like the fault modeling aspect.

In a leaf in the process hierarchy only the input and output signals are specified. The graphical editor allows, at any time, extension of the process model to greater detail or modification of a process as long as the interface to process remains the same or is extended. Various popup menus allow the user to specify new signals, subprocesses and connections between them. In order to represent subprocesses graphically the user can use a bit map editor to create an icon for a specific process. Each icon also contains a connection point for each input and output signal in order to enable the user to specify connections to and from those signals on a higher level.

5.4.4 Hierarchical Fault Models (HFM) for the ECLSS/PWP

Once the process model is defined, different views of the processes can be studied and modeled. In this section we are considering a fault model for the ECLSS. We would like to emphasize that the fault specification is incomplete and needs further refinements.

The construction of a fault model is performed the same way as that of a process model, namely using an appropriate graphical editor. Each fault model is represented on the higher level of the hierarchy by an icon where each icon has several connection points related to the failure modes of the subprocess. An important aspect of fault modeling is its close relationship to the process models. When a fault model for a process is to be defined, the fault model will *inherit* the basic structure from the HPM: (1) the name of the subprocesses, and (2) the causal links among the the subprocesses which is derived from

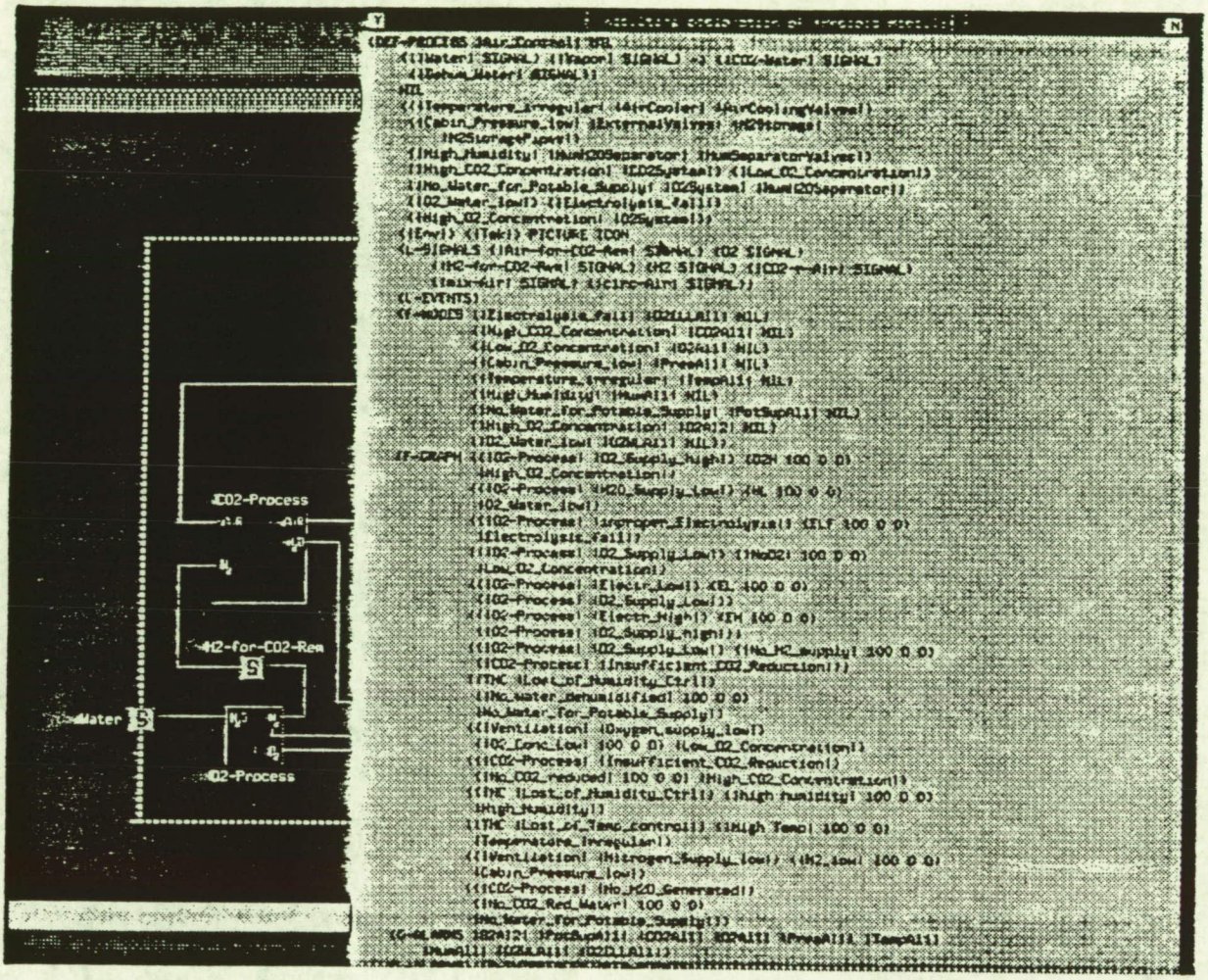

Figure 5.10: Declarative form of the AirControl Process

the existence of physical links. This relationship guarantees that the HPM and HFM will be consistent.

In the following, some of the failure modes and their relationships will be explained.It does not have any failure modes but defines possible relationships of failure modes of its submodels.

As mentioned earlier the fault model is one aspect of a process model besides the structural view. Therefore the fault model is also stored in the declarative form of the process model. This is shown in Figure 5.11 for the toplevel process. The declarative form holds structural and fault model information about the process in different view slots, which are both accessed by the diagnosis system.

5.4.5 Definition of repair actions

The most obvious place for incorporating repair action definitions is the HPCM, where the physical components are enumerated together with their fault modes. Unfortunately, repair actions cannot be incorporated into HPCM for the following reasons: HPCM is built *incrementally*, and each declaration describes a physical component *type*, and *not* a real, component *instance*. For this reason we have chosen a different method.

If we want to coupld the physical component hierarchy to the structural, geometrical simulation, we need to solve two problems:

1. Identification of geometrical structures with abstract physical components in the HPCM.

2. Assignment of repair actions to physical component nodes in the HPCM.

The most plausible way for identifying an object (a node) in a hierarchy is the describe the path which leads to it, starting from the toplevel. We have used this method the following way: To identify the physical component with a geometrical object we introduced the following declaration:

```
(def-physical-names
  {({PathToPhysicalComponent} GeometricalObjectName ) }*
)
```

The `PathToPhysicalComponent` is a list of symbols which lead to the desired physical component in the hierarchy, while `GeometricObjectName` is a name from the geometrical modeling environment. (It is usually an argument in one of the `make-object` commands, described above.)

To facilitate the assignment of repair actions to physical components, the following declaration can be used:

```
(def-repair-actions
  {({PathToPhysicalComponent} RepairActionString ) }*
)
```

C-2

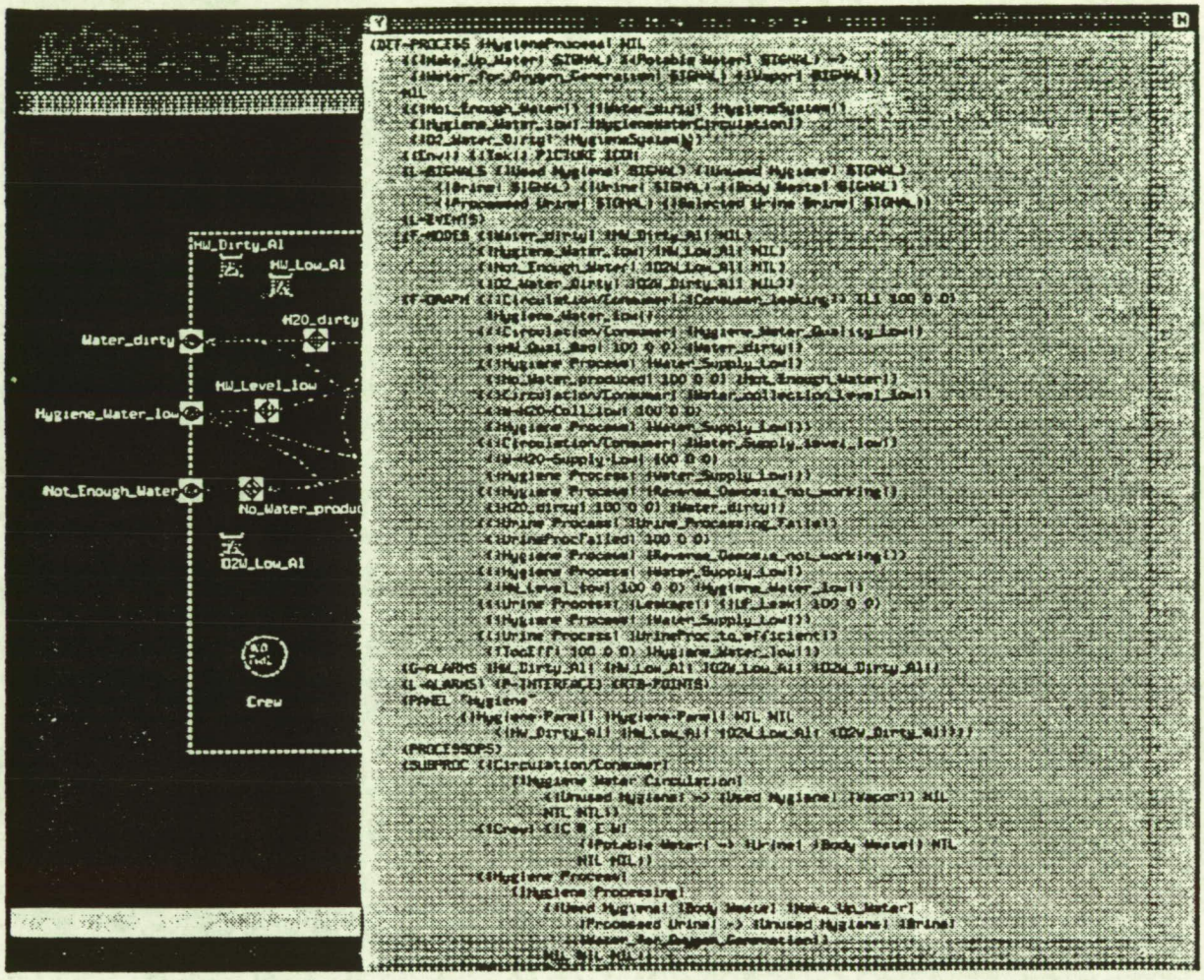

Figure 5.11: Fault Diagnosis Declarative Form

Its is similar to the previous declaration, but `RepairActionString` denotes the repair action to be executed when the diagnostics initiates a fault recovery.

5.4.6 Integrated monitoring and diagnostics with robot simulation

After completing the HPM and HFM for the ECLSS/PWP, we prepared a demonstration which integrated the monitoring and diagnostics system (built using MPE) with the geometrical modeling environment. The integration was realized with the help of pipes, a standard Unix interprocess communication facility.

The integrated system works the following way: The monitoring and diagnostics system is loaded with the symbolic model sets of the ECLSS/PWP. The model set also contains the physical component/geometrical object associations, as well as the definitions of various repair actions. The repair actions are simply the names of command files, executable by the geometric modeler. When the monitoring a diagnostics system is built up, it starts the geometric modeling environment (which is just another Unix program), and initializes it to show the starting configuration of the ECLSS/PWP. A fault-simulation (an automatic scenario generator) is also created, which is responsible for creating alarm sequences for exercising the diagnostic system. The operator can select a physical component for fault, and the program starts generating the alarm sequence which corresponds to the one in a real-life system. The diagnostics system analyzes the alarms and tries to pinpoint the fault source: a physical component. When it found one it indicates that by coloring the corresponding *component button* on the operator panel of the diagnostics to red, and by coloring the corresponding *geometrical object* to red. Finally, if there is an associated repair action defined, it sends the commands of it to the geometric modeler for execution. The robot simulator then shows the execution of the repair action, as a three-dimensional animation. (See Fig 5.12.)

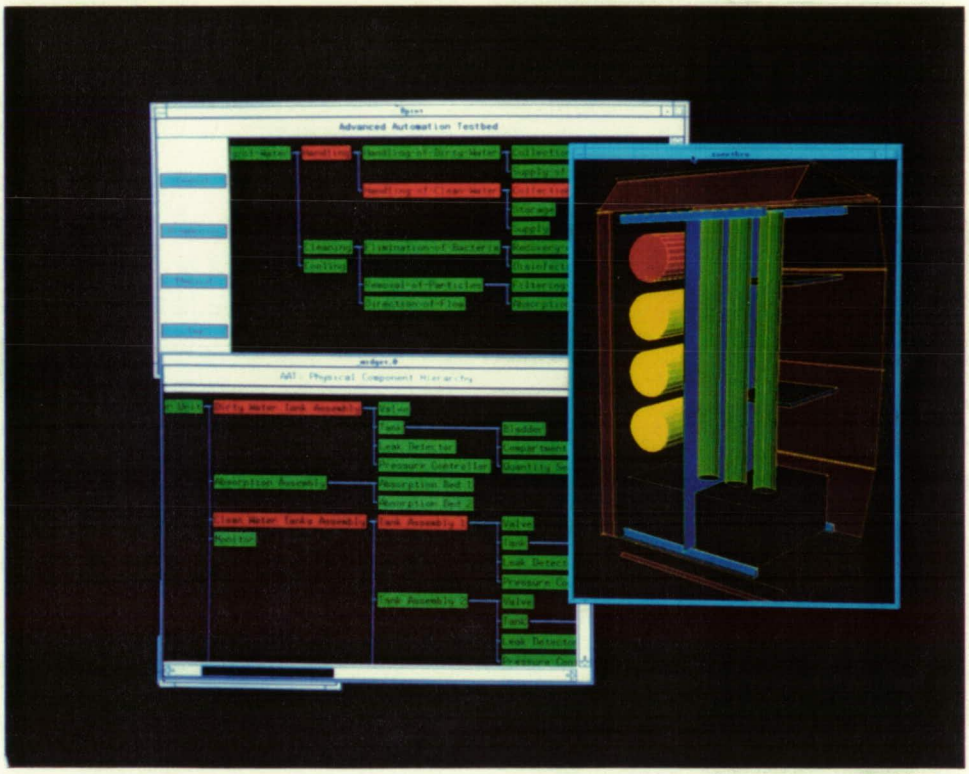

Figure 5.12: Integrated 3D models and fault monitoring/repair system

Chapter 6

Suggestions for Future Work

During this project we have obtained experiences with the potential of a graphic workstation environment in robot simulation and have tested the new capabilities of the AI extension of ROBOSIM. The research has been conducted in three parallel directions:

- Improvement of the basic capabilities of ROBOSIM by taking advantage of the graphic workstation environment.
- Integration of ROBOSIM with model–based modeling techniques and extension of the capabilities of the basic system with support toward general automation problems.
- Continuous testing of the system with a variety of application problems.

The results of the project can be summarized as follows:

- ROBOSIM is available now on different graphics workstations, including HP 9000 Series and Intergraph.
- ROBOSIM has been extended with:
  - Collision detection
  - 3D model editing facilities
- An extended modeling environment (*Agent*) has been built using ROBOSIM: it is upward compatible with the new package.
- The graphical modeling environment has been coupled with a *model-based* system for describing and simulating robot controllers.
- Case studies have been performed using the tools developed in this project. The studies include: (1) geometrical modeling of the Space Station, (2) process modeling of the Environmental Control and Life Support System, which make possible integrated fault diagnostics and fault recovery, the latter being simulated using the 3D graphical robot simulation package.

There are many directions in which this research can be extended.

- *3D graphical editing tools.* ROBOSIM has a rudimentary facility for creating robots and geometrical structures interactively, and it can be extended into a sophisticated tool for editing 3D models. It is a very interesting and promising research area: currently there are no (commercial) systems which support 3D editing. The potential benefit of this project would be the development of user interface methods for interactive 3D manipulation.

- *ROBOSIM/CAD interface.* There are many cases when ROBOSIM must use geometrical models available in another environment, e.g. an engineering CAD system. This system would serve as a generic interface between the ROBOSIM modeling language and CAD systems, so that the 3D models of a CAD system can be incorporated into robot simulations automatically. The result of this research would be a translator program which converts CAD files into ROBOSIM models.

- *ROBOSIM/CLIPS interface.* NASA is widely using the expert system shell CLIPS. It would be very advantageous to provide a communication mechanism between ROBOSIM and CLIPS, and make possible the integration of robot simulations with CLIPS-based applications. Robot planning systems can easily be implemented in CLIPS and an integration of ROBOSIM with CLIPS would make possible to extend already existing CLIPS applications with a 3D graphics output facility.

- *Integrated modeling environment for automation systems.* Currently, ROBOSIM is used for geometrical modeling and there are other symbolic tools for creating abstract process models, etc. An integrated environment would contain *both* models in one integrated formalisms. This would facilitate a more efficient representation, where the abstract (logical) models and the concrete (geometrical) models of automation systems can be stored together. An integrated modeling environment like this can serve as an example system for a sophisticated automation development environment.

Appendix A

Structure Declarations for the Simulation Library

This Appendix contains all structure declarations used throughout the code of the Simulation Library. The declarations are given using the conventions of the C programming language, since the Simulation Library itself was coded in C.

```
typedef float  (*S_VECTOR)[3];

struct s_penalty {
    float pen_norm[3];
    float pen_dist;
};

typedef struct s_poly {
    float norm[3];
    float nd;
    int vec_ptr;
    int num_vectors;
    struct s_penalty *pen;
} *S_POLY;

typedef struct s_link{
    int num_vectors;
    S_VECTOR list_ptr;
    float *md;
    int display_list;
    float bbc[4]; /*bounding box center*/
    float bbd[4]; /*bounding box half-diagonal*/
    float INERT[4][4];
    float CURR[4][4];
```

```
    int  num_poly;
    S_POLY  poly;
    float theta,dz,da,alpha;
    int jtype1,jtype2;
    float JNT1[4][4];
    float JNT2[4][4];
    float AMAT[4][4];
    float TRANS[4][4];
    float curr_var;
    float min_var;
    float max_var;
} *S_LINK;

struct s_robot{
    int num_links;
    S_LINK link[18];
    float Pre[4][4];
    float Post[4][4];
    float DH[18][5];
    int display_list;
    float POS[4][4]; /*matrix describing position of robot in environ*/
    int (*INV_KIN)(); /* pointer to function that solves inverse kin */
};

struct s_env{
    int num_vectors;
    S_VECTOR list_ptr;
    float *md;
    int display_list;
    float bbc[4]; /*bounding box center*/
    float bbd[4]; /*bounding box half-diagonal*/
    float INERT[4][4];
    float POS[4][4];
    int  num_poly;
    S_POLY  poly;
};

/* this is a copy of s_env, however it is also the generic type */
/* of which link and obj can be cast into */
struct s_gen{
    int num_vectors;
    S_VECTOR list_ptr;
```

```
    float *md;
    int display_list;
    float bbc[4]; /*bounding box center*/
    float bbd[4]; /*bounding box half-diagonal*/
    float INERT[4][4];
    float POS[4][4];
    int  num_poly;
    S_POLY  poly;
};

struct s_obj {
    int num_vectors;
    S_VECTOR list_ptr;
    float *md;
    int display_list;
    float bbc[4]; /*bounding box center*/
    float bbd[4]; /*bounding box half-diagonal*/
    float INERT[4][4];
    float POS[4][4];
    int  num_poly;
    S_POLY  poly;
    float DIFF[4][4];
};

typedef struct s_any {
    int     type; /* 0=robot 1=env 2=obj */
    int     in_use;
    union {
struct s_robot *r;
struct s_env *e;
struct s_obj *o;
} obj;
} *S_ROBOT,*S_ENV,*S_OBJ;

typedef struct s_list {
    struct s_any *item;
    struct s_list *next;
} S_LIST;

typedef struct s_collision{
    struct s_any *S1;
    int L1;
```

```
    struct s_any *S2;
    int L2;
} S_COLLISION;

typedef float S_POINT[3]; /* X Y Z */

typedef float S_ORIENTATION[3]; /* ROLL PITCH YAW */

typedef float S_LOCATION[6]; /* X Y Z ROLL PITCH YAW */

S_ROBOT S_GET_ROBOT();
S_ENV S_GET_ENV();
S_OBJ S_GET_OBJ();
typedef float S_JOINT[18];

typedef float matrix3d[4][4];
struct matrix_struct {matrix3d msxform; };

#define S_REPLACE_MATRIX_3D(dest,src) \
    *(struct matrix_struct *)(dest) = *(struct matrix_struct *)(src)
```

Appendix B

Simulation Library Functions

This Appendix contains the interface declarations to the functions of the Simulation Library. The declarations are given using the conventions of the C programming language, since the Simulation Library itself was coded in C.

```
S_GET_ROBOT (filename)
    char *filename;

S_GET_ENV (filename)
    char *filename;

S_GET_OBJ (filename)
    char *filename;

S_PRETRANSLATE (o,x,y,z)
    S_ANY o;
    float x,y,z;

S_POSTTRANSLATE (o,x,y,z)
    S_ANY o;
    float x,y,z;

S_PREROTATE (o,x,y,z)
    S_ANY o;
    float x,y,z;

S_POSTROTATE (o,x,y,z)
    S_ANY o;
    float x,y,z;

S_MOVEJ (r,joints)
```

```
    S_ROBOT r;
    S_JOINT joints;

S_CLEAR_JOINT (joints)
    S_JOINT joints;

S_MOVEJI (r,joints,steps)
    S_ROBOT r;
    S_JOINT joints;
    int steps;

S_USE (o)
    S_ANY o;

S_DONTUSE (o)
    S_ANY o;

S_CHECK (o)
    S_ANY o;

S_CHECK_ROBOT (r)
    S_ANY o;

S_C_SWITCH (x)
    int x;

S_COLLIDE ()

S_SET_INV (r, inv_func_ptr)
    S_ROBOT r;
    int (*inv_func_ptr)();

S_KINV(r, dlm, joints, reset)
    S_ROBOT r;
    float dlm[4][4];
    S_JOINT joints;
    int reset;

S_JACOB(joints, jac)
    S_JOINT joints;
    float jac[6][6];
```

```
s_translate (mat,x,y,z)
    float mat[4][4],x,y,z;

s_rotate (mat,x,y,z)
    float mat[4][4],x,y,z;

s_rotatez (mat,z)
    float mat[4][4],z;

s_transpose (mat)
    float mat[4][4];

s_invert (mat)
    float mat[4][4];
```

Appendix C

HDL/C Interface

C.1 HDL/C Interface

C.1.1 HDL Parameters

In HDL there are two kinds of parameters available: static and dynamic. *Static* parameters are determined at the time when the HDL declaration is instantiated and they don't change their value. Static parameters are passed *by value* to HDL components. *Dynamic parameters* are initialized at instantiation time, but they can change their value as the Multigraph network is running. Dynamic parameters are passed *by reference* to HDL components.

Both kind of parameters can have two attributes: (1) default value, and (2) type. At editing time these can be specified using the "Modify" command of the graphical editor and clicking on the appropriate parameter. The default value can be anything, but it *must* be compatible with the type. There are four possible types for a parameter: (1) `integer`, (2) `float`, (3) `double`, (4) `pointer`, and (5) `lisp`. In general, the type for parameters must always be specified, the specification of default value is optional.

In HDL all the low-level algorithms should be embedded in *primitive* declarations which specify a simple algorithm and its interfaces to the Multigraph network and the parameters. An HDL primitive declaration has the following form:

```
(defprimitive
 Name Discipline
 Io_interface Parameter_interface Control_interface
 Environment_interface Task_interface
 Icon Picture
 Body)
```

The `Body` component of the declaration can have one of the following format:

- Lisp expression. In this case it is considered as the body of the Lisp function which implements the algorithm.

- A string[1] which contains one name. In this case it is considered as the name of a C function which implements the algorithm. The C function should be loaded explicitly before the HDL interpreter processes the primitive declaration.
- A string which starts with a { character and contains the C function body. It must end with a }, as the syntax rules of C dictate. In this case the C function text is embedded in the declaration, and the HDL interpreter will compile and load this C text whenever the primitive is encountered for the first time.

In the subsequent sections the methods of script writing are reviewed.

C.1.2 HDL Context Tables

When the HDL interpreter is instantiating a primitive object it constructs a context table *from* the actual parameter values of the object and *according to* the formal parameter specification in the declaration.

The static parameters are always passed by values. Dynamic parameters are *encapsulated* in a special structure and this structure is passed to all primitives which refer to the same parameter. If the dynamic parameter was specified of `integer`, `float`, `double`, or `pointer` a Multigraph buffer[2] is created which is large enough to hold data and the address of this block is passed. If the dynamic parameter is of `lisp` type a Lisp structure is created which can hold the data.

The context table is built from static parameter values and dynamic parameter references as follows. When the script of the primitive is a Lisp script it simply creates a Lisp vector which contains first the static parameter values, then the references to the Lisp objects keeping the dynamic parameters, *in the order they appeared in the primitive declaration.* The HDL interpreter also builds macros for referring to input and output port names and parameters by their *name.* These macros surround the Lisp script at execution time, so the script can access its parameters by simply referring to them by name. It is recommended that for Lisp scripts only parameters with type `:lisp` be used.

When the script is written in C the interpreter constructs a context table from the values and references, in the order they appeared in the declaration. The interpreter then passes the address of this table to the actornode which runs the script. C scripts can have (static or dynamic) parameters with types `integer`, `float`, `double`, or `pointer` *only.*

Example: Suppose we have defined a primitive with the following static parameters:

```
(|Ts| 0 :INT)
(|Kp| 0.0 :DOUBLE)
(|Kd| 0.0 :FLOAT)
(|Ki| 0.0 :POINTER)
```

and with the following dynamic parameters:

[1]Strings in Lisp are sequences of characters surrounded by double quotes: ".

[2]An MG buffer is a dynamically allocated block of raw memory.

```
  (|P_result| 0.0 :DOUBLE)
  (|P_error| 0.0 :DOUBLE)
  (|P_P_error| 0.0 :DOUBLE)
```

The HDL interpreter will construct a context table which is equivalent to the following C structure:

```
struct {
  int Ts;
  double Kp;
  float  Kd;
  void *Ki;
  double *P_result;
  double *P_error;
  double *P_P_error;
}
```

As it was mentioned above, there are two ways for referring to C scripts: (1) by name (the "plain" method), and (2) by inserting the entire script in the declaration (the "embedded" method). In the following section the methods for preparing these C scripts are described.

C.1.3 Preparing and loading "Plain" C scripts

In the case of the plain method the declaration of the primitive contains only the name of the C function which implements the script. In this case it is the user's responsibility to create write, compile and load a C file which contains the script. The file can contain many scripts, as well as other C functions used by those scripts. The file should have the following general structure:

```
/* Include MGK extern functions and macros: */
#include "cmgkdefs.h"
/* Other includes */
...
/* Forward declaration of functions: */
void UserFunction1Name();
...
/* First mandatory declaration in the file: */
make_binder(init_name)
  /* Bind user function "UserFunction1Name" */
  bind(UserFunction1Name)
  /* Bind user function "UserFunction2Name" */
  bind(UserFunction2Name)
```

```
  /* Other bindings: */
  ...
end_bind(Initialization)

/* Usual C code comes here */
...
void UserFunction1Name(cntx)
struct { ... } *cntx;
{
 ...
}
...
```

The file "cmgkdefs.h" contains the MGK macros (see [2]) necessary for correct compilation of the file. The "substantial" part of the file has to begin with the mandatory `make_binder`, `bind`, `end_bind` construct. These are three C macros which together generate a special C function that initializes the internal tables of MGK when the file is loaded into the MGK run–time environment. The name of this C function is `init_name`, what the user should specify.

The user functions to be used as scripts for HDL primitives should be placed into the internal tables of MGK by using the `bind` macro: the name of the C function must be the only argument for this macro. Note that for C the function name should be forward declared *before* referring to it. The `end_bind` macro closes the `start_bind` construct; the user can specify optional initialization code here as the argument for the `end_bind` macro. If there is nothing to be initialized, the macro should be called with an empty argument list (i.e. with "()").

Once the C file is prepared it must be compiled as follows:

```
cc -IMGKDIR -c file.c
echo '#()' | cat >>file.o
```

`MGKDIR` should specify the directory where the "cmgkdefs.h" file is to be found (if it is not the current directory). The second command line appends a special marker to the end of the file for the KCL loader. It is not necessary for all the C scripts to be stored in one file, but there must be one and only one file that contains the `start_bind` macros as above. The other files should be compiled using the usual method of C compilation for creating a C relocatable object file [3]

Once the files are compiled they must be loaded into the KCL/HDL environment. Suppose the main C file is called `foo.o`, there are two auxiliary files `bar.o` and `baz.o`, and we also want to load the math library of C. There are two equivalent ways of doing this:

- Using the KCL "faslink" feature:

[3] Use the `-c` flag.

```
(si:faslink "foo.o" "bar.o baz.o -lm")
```

- Using the MGK loader:

```
(load-modules "foo.o" :linkeropts "bar.o baz.o -lm")
```

In both cases it is supposed that MGK has been initialized, and in the second case that the "MGK" package is in use. The second way is preferable if one wants to load in scripts into remote MGK tasks, because a task can also be specified as an optional parameter (see [2]).

After the C object files have been prepared, compiled and loaded as described above they are ready for use in HDL structures. Note that the names of entry points specified in the `bind` macro and in the HDL primitives should exactly match, and the object modules should be loaded before starting the building of HDL structures.

For a C script one must declare an appropriate context structure, the address of which is going to be the only parameter for the function. It is the user's task to exactly match with the datastructures created by the HDL interpreter, as described above. It is also the user's task to refer to the ports of the actornode which runs the script by the appropriate indices. The input and output ports of the actornode are identified by an index (starting from zero), in the order they appear in the declaration.

An example for a plain C script and its corresponding HDL declaration may look like this:

```
;;; HDL primitive:
(defprimitive TActor :ifany
   ;; One input, one output:
   (In -> Out)
   ;; One static parameter, one dynamic:
   ((Add 0 :int)) ((Mult 0.0 :double))
   ... etc.
   "TestActor")

/* C file containing "TestActor" */
#include "cmgkdefs.h"

/* Context table structure: */
struct Context {
  /* Static parameter: */
  int add;
  /* Dynamic parameter: */
  double *mult;
};
```

```
/* Forward declaration: */
void TestActor();

/* Mandatory code: */
start_bind
  bind(TestActor)
end_bind()

/* Code for the primitive: */
void TestActor(cntx)
struct Context *cntx;
{
  double data,newdata;
  /* Receive DOUBLE data: */
  data = mgk_d_receive(0);
  /* Multiply it with the dynamic parameter
     and add the static parameter to the result:*/
  newdata = data*(*(cntx->mult))+cntx->add;
  /* Send the result: */
  mgk_d_propagate(0,newdata);
}
```

C.1.4 Preparing "Embedded" C scripts

When embedded scripts are used the full script text is to be typed in the declaration. It should be in the form of a Lisp string amd it *must* start and end with the characters { and }, respectively. The Lisp string facility imposes certain restrictions on the contents of the script: quotation marks are to be escaped, (i.e. a quation mark in the string should be written as \"), and the escape character \ is to be escaped with itself.

To use embedded C scripts offers significant advantages: the HDL interpreter generates accessor macros for both the actornodes ports and context elements, and it compiles and loads the script automatically. The user can refer to the ports and context elements (i.e. the parameters) by their name. Dynamic parameters are passed by reference, therefore the user *must* dereference them if he wants to get their value. And example HDL script with embedded C code may look like this: (Note the use of escape characters which are mandated by the Lisp strings.)

```
(DEFPRIMITIVE PID-PRIM :IFALL
 ((SP :STREAM) (MV :STREAM) -> (CS :STREAM))
 ((|Ts| 0 :INT) (|Kp| 0.0 :DOUBLE) (|Kd| 0.0 :DOUBLE)
  (|Ki| 0.0 :DOUBLE))
 ((|P_result| 0.0 :DOUBLE) (|P_error| 0.0 :DOUBLE)
  (|P_P_error| 0.0 :DOUBLE))
```

```
...
"{
  double sp = mgk_d_receive(SP);
  double mv = mgk_d_receive(MV);
  double error = sp - mv;
  double result;
  printf(\"SP = %d\\n\", SP);
  printf(\"MV = %d\\n\", MV);
  printf(\"CS = %d\\n\", CS);
  printf(\"TS = %d\\nKp = %f\\nKi = %f\\nKd = %f\\n\", Ts, Kp, Ki, Kd);
  printf(\"P_result = 0x%x\\nP_error = 0x%x\\nP_P_error = 0x%x\\n\",
      P_result, P_error, P_P_error);
  printf(\"*P_result = %f\\n*P_error = %f\\n*P_P_error = %f\\n\",
      *P_result, *P_error, *P_P_error);
  printf(\"\\n\\n\");
  result = *P_result +
               Kp * ( (error - *P_error)
                    +(Ts * error * Ki)
                        +(Kd/Ts)*(error - 2* *P_error + *P_P_error)
                   );
  *P_P_error = *P_error;
  *P_error   = error;
  *P_result  = result;
  mgk_d_propagate(CS,result);
}")
```

Note that the names of the parameters are the same as in the model, but they do not have the special Lisp marker (the | character) around them.

When the HDL interpreter is instantiating a primitive with an embedded script, it checks whether the script is compiled and loaded yet. If not it creates a C file from the script and calls the compiler on it. The above example has the following corresponding C file:

```
#include "cmgkdefs.h"
void HdlPIDPRIM();
make_binder(init_HdlPIDPRIM)
bind(HdlPIDPRIM)
end_bind(;)
typedef struct {
int _Ts;
double _Kp;
double _Kd;
double _Ki;
```

```
double *_P_result;
double *_P_error;
double *_P_P_error;
} HdlPIDPRIM_cntx;
#define SP 0
#define MV 1
#define CS 0
#define Ts (_cntx->_Ts)
#define Kp (_cntx->_Kp)
#define Kd (_cntx->_Kd)
#define Ki (_cntx->_Ki)
#define P_result (_cntx->_P_result)
#define P_error (_cntx->_P_error)
#define P_P_error (_cntx->_P_P_error)

void HdlPIDPRIM (_cntx)
HdlPIDPRIM_cntx *_cntx;
#line 1
{
  double sp = mgk_d_receive(SP);
  double mv = mgk_d_receive(MV);
  double error = sp - mv;
  double result;
  printf("SP = %d\n", SP);
  printf("MV = %d\n", MV);
  printf("CS = %d\n", CS);
  printf("TS = %d\nKp = %f\nKi = %f\nKd = %f\n", Ts, Kp, Ki, Kd);
  printf("P_result = 0x%x\nP_error = 0x%x\nP_P_error = 0x%x\n",
      P_result, P_error, P_P_error);
  printf("\n\n");
  result = *P_result +
              Kp * ( (error - *P_error)
                    +(Ts * error * Ki)
                        +(Kd/Ts)*(error - 2* *P_error + *P_P_error)
                   );
  *P_P_error = *P_error;
  *P_error   = error;
  *P_result  = result;
  mgk_d_propagate(CS,result);
}
```

The user can force the compilation and loading of all the embedded scripts by calling the built-in function: `(hdl-compile-scripts)`

There are a couple of utility macros which can be used for controlling the compilation and loading.

- `(def-mgk-library "MGKLIBRARYPATH")`
  This macro defines where the MGK macro file `cmgkdefs.h` is located. It must be set correctly before attempting any embedded script compilation. It defaults to `"."`.

- `(def-hdl-library "HDLLOADLIBRARY")`
  This macro defines what second argument string is to be passed to the loader when the compiled script is loaded. The user can specify in this string arbitrary object file names and object library names. It defaults to `"-lc"` (the C runtime library).

- `(add-hdl-library "HDLLOADLIBRARY")`
  This macro prepends its argument string to the current value of loader argument string specified in the `(def-hdl-library)` macro.

- `(add-include-library "INCLUDELIBRARY")`
  This macro is used to specify additional files for including at the time of compilation of the script. Thd HDL interpreter generates lines of the following format: `# include "INCLUDELIBRARY"`. The user place the file name in the proper quotes, i.e. the argument string should look like as `"\"filename\""`, or `"<filename>"`.

When the HDL interpreter runs the C compiler it sets the line counter of the compiler to the first line of the script. Therefore, the C compiler error messages will refer to lines relative to the first line of the embedded script. Unfortunately, there is no debugging facility for the embedded scripts, thus the users are cautioned to take extreme care in developing these scripts.

Bibliography

[1] Sztipanovits, J.: *MULTIGRAPH: Architecture for Intelligent Measurement and Control Systems (Introduction and Overview).* Department of Electrical Engineering, Vanderbilt University, 1986.

[2] Biegl, Cs.: *Multigraph Kernel Interface Description, User's Manual.* Department of Electrical Engineering, Vanderbilt University, 1986.

[3] Springfield,J.: *ROBOSIM Workstation Extensions*, M.Sc. Thesis, Vanderbilt University, 1988.

[4] Springfield,J., Cook, G.E., Andersen, K., and Fernandez, K.R.: *ROBOSIM: A Simulation Package for Robots*, University Programs in Computer-Aided Engineering, Design, and Manufacturing, ASCE, 1989.

[5] Walter,S.E. *Polygonal Collision Detection Algorithm*, Ph.D. Dissertation, Cornell University, 1985.

[6] Wilson,S.L. *Interfacing of a Robot Simulation Package with Graphics Utilities of an Intergraph Interpro 360 System*, M.Sc. Thesis, Vanderbilt University, 1990.