SEFFAS
Simulator Environment For Flexible Assembly Systems

A.H.J. Koning

April 1991
Abstract

This paper describes the design and implementation of a simulator for so-called flexible assembly systems. These are robot cells with multiple robots and sensors designed to be able to produce different kinds of products. This project is a part of the Esprit Computer Integrated Manufacturing Project No. 2202 - PLAT - Planning Toolbox for CIM systems. The simulator has to be capable of interactive simulating a robot cell with multiple robots and sensors. With simulating we mean both the emulation of the control of the cell, and the visualization of the simulated robot cell on workstations, so that we can monitor the (emulated) execution of commands and programs by the cell.
Contents
1 Introduction

1.1 Background

With the evolution of the electronic computer, which started in the early 1950's, came the evolution of the robot in the early 1960's. But it was not until the late 1970's that robots became cheap and versatile enough to be used on a large scale in the industry. During the years that followed robot utilization increased rapidly and in the mid 1980's it was expected that automated assembly systems (i.e., robots) would replace the traditional systems before 1991. But contrary to the expectations in the second half of the 1980's the increase in robot utilization came to a halt. The reason for this was that although the mechanical evolution had led to robots capable of almost any manipulation needed, the real problems lay with the software development and the integration and interpretation of sensor data in the assembly process. Research in these areas has led to the insight that robots must not be seen as isolated units but that for successful application of robots it is essential that more emphasis is put on the integration aspects, leading to the overall concept of computer integrated manufacturing (CIM). Together with the introduction of the robot came the problem of how to program it. Early techniques were based on the teach-in concept. The programmer physically moves the robot through the required trajectory during which the motions of the robot's joints are recorded on tape or otherwise. This recording then serves as the robot program. Drawbacks of this method are that the robot has to be taken out of the production process to be taught and that there is little control over the correctness of the program. With the introduction of textual representation of robot motion the second problem was tackled and the first robot control languages were developed. These first languages were very simple and with the development of high-level general purpose languages such as C and Pascal came more and more complex robot languages, some of them based on extensions or adaptations of general purpose languages. This however led to the problem that programming the robot was no longer a job for the unskilled operator but that specially trained programmers were needed. Nevertheless at the moment most industrial robots come with an interpreter for a specialized robot language. However this still doesn't solve the problem of having to take the robot out of the production process to program it. This programming with the help of robot's controller itself is called on-line programming and to overcome this problem of loss of productivity off-line programming is introduced. Off-line programming allows us to write the program on a other system than the actual robot and thus is very appealing [Gini]. The idea of an off-line programming system is that we can test the correctness of our program without having to use the real robot. To be able to do this the off-line programming system should contain a model of the robot and its environment. The model should correspond closely to the real situation in order to ensure that the program functions correctly.
into CIM" was the development of tools for production planning and off-line programming. Production planning and programming in Computer Integrated Manufacturing are much more complicated problems than the off-line programming of a single robot. This means that there is a greater chance of errors in the planning and/or programming than with a single robot. So in addition to the reasons mentioned above for a single robot, the fact that an error in the planning or programming of a CIM system can lead to the breakdown of the complete production in a factory makes that we would like to be able to simulate our CIM system so that we can test our planning and programs before they are executed.

Among the tools developed as part of this ESPRIT project is the off-line programming and simulation package ROSI [Dillmann][Huck][Rembold] developed by the University of Karlsruhe. ROSI enables the modelling and programming of robots, parts and the robot's work cell. Part of ROSI is the simulation of the robot trajectories by a graphics system. In addition to programming and planning errors, it is possible that changes in the actual status of the robot's environment may lead to interruptions or even a breakdown of the production process. This is because in the planning and programming stage we assume a certain model of the reality, which may only partially reflect the real world. To prevent this from happening we want our robot programs to be able to handle these so-called exceptions. A robot programming environment capable of handling exceptions has been proposed [Meijer1] and implemented [Mul]. The University of Amsterdam participated in this ESPRIT project with the development of the "Exception Handling Model" (EHM). Exception handling in this context means that the robot is able to react to unforeseen situations in such a way that they can be handled adequately without outside help, so that the robot system continues to operate. Tools have been developed to generate and monitor the recovery procedures. One of these tools [Meijer2] uses an expert system to implement exception handling. The development and testing of this tool was done with a real robot and real sensors. The drawbacks of this approach are obvious: You can only test the system with the available sensors and robots, preventing them from being used for other purposes and until the exception handling is perfect it is possible for things to go wrong and the set-up may get damaged. A suitable simulator would remove these drawbacks. Work along these lines currently continues in Espirit II project 2202: PLATO - Planning Toolbox for CIM systems. This project has as one of its objectives the development of a complete simulation system for CIM systems. The University of Amsterdam will provide sensor-based control and exception handling in this system. Again a simulator capable of handling multiple robots and sensors is needed to create a suitable programming and testing environment.

1.2 Demands

The availability of low-cost Unix workstations with bit-map graphics displays opened up a new area in off-line robot programming. These workstations are
well suited for the off-line programming of robots and when placed in a network these workstations allow companies to have their departments do product development and production planning at various sites. Research and development is now going on in this field of application, for instance the ESPRIT PLATO project. As a part of this ESPRIT project this graduation project involves the design and implementation of a simulator system suitable to run interactively on Unix workstations, and Sun workstations to be specific, and capable of giving a detailed simulation of an arbitrary robot cell including the integration of sensors. As an example of this sensor integration we can take two robots in a master-slave configuration, in which one follows the other based on information from a force sensor held by both robots. For our simulator we would like to use a standard windows environment and a standard graphics library so that the simulator package can easily be ported and made available on a large number of machines. As we already mentioned Unix is the most widely spread operating system for workstations these days and for Unix systems is the X-windows environment expected to become the standard window environment. The PHIGS graphics library is the standard for this type of visualization. So what we ultimately would like is to have our simulator run under X-windows and making use of the PHIGS library.

1.3 Outset

At the start of this graduation project we had the result of two previous projects at our disposal [Roth][Mul]. This simulator called ROS or ROSI was a partial port of the ROSI package mentioned earlier, designed to run on Sun 3 workstations, and it allowed us to simulate multiple robots. It used the PHIGS library and ran under the SunView window environment. But while it fulfilled part of our demands it failed in two important areas: It took the simulator 20 seconds or more to draw one robot, making it more or less impossible to be used interactively. Furthermore the simulator made no provision for the integration of sensors and neither was there a way to access the world model used from outside the simulator. Although this simulator did not run under the X-Windows environment, using a PHIGS library suitable for X-windows would enable it to do so. We started this project by looking if it was possible to use the existing simulator and try to find ways to speed it up (chapter 2). When we received a new and considerably faster version of this simulator we decided to use it and next we concentrated on integrating the various parts of our system which we already had available (chapter 3-4) and extending and changing them where necessary (chapter 4-5). This led to a simulation environment which allowed us to realize an interactive simulation of multiple robots. The important part, sensor integration, was next and we designed and implemented a way to be able to do this (chapter 6-7). Finally we look at the result of this project and to future developments (chapter 8).
2 Orientation

2.1 First Effort

The simulator we had at our disposal was written in Pascal and used an implementation of the PHIGS standard graphics library written in FORTRAN. As we mentioned earlier it ran on Sun 3 workstations. Because we now also had the newer and considerably faster Sun 4 workstations to work with it seemed to be a good idea to try and port ROSI to these machines to see what increase in speed this would give us. Despite repeated efforts we did not succeed in porting the PHIGS graphics library (FIGARO), so we could not run our simulator on the faster SUN 4, SPARC station 1 and SPARC station 1+ machines we had available. And while the PHIGS package was a commercial product it did not integrate well with the SunView windows environment it was running under, so one of our options was to replace it. We took the picture quality of the ROSI simulator as our minimum target and searched for a way to get the performance we required to do real-time animation of the simulated robot cell. This led to the following experiment:

2.2 Experiment

The first question we had to answer was whether the poor performance of the existing program was a software or a hardware problem. We had little documentation about the way PHIGS works, so this was one of the first things to figure out. PHIGS uses an hierarchic tree structure to store the transformation on the objects it has to draw and keep pointers to an array with the geometric data. When given an order to draw an object it walks through the tree, performs the transformations and draws the object. This implies that either the transformations or the drawing or both are costly operations in terms of processor time. To find out where the problems really lie we did some experiments. First we needed an estimate of the complexity of the picture we want to draw. Because we use vector graphics the number of edges in the picture will provide such an estimate. When we looked at the data files used by the existing program to describe the various elements in our picture, we saw that they consisted only of polygon descriptions. One line to describe the polygon, followed by the vertices of that polygon, with each vertex on a separate line. Since the number of vertices of a polygon is the same as the number of edges of that polygon, we can use the length of the file in lines as the upper limit of the number of edges in the picture. This yields the following results: About 3500 lines (edges) for a Puma 260 robot, and about 4000 for a Bosch robot. Since our application will be used to simulate cooperating robots we are looking at a picture with at least about 7,000-8,000 lines. And as we want an animated picture we probably need to redraw the picture at least twice per second. This brings us to a total of 15,000 lines a second. The transformations are done by means of floating point matrix-vector multiplication.
s, so we wrote a small program that does 15,000 of those multiplications and timed it. It took a Sun 3 less than a second to execute, so we can more or less safely assume that this is not the bottleneck in our application and that the lack of performance is caused by the limited speed with which the lines are actually drawn. To test this assumption we wrote and timed a little program which only draws lines. Since most lines in our application are pretty short, our benchmark program drew 15,000 lines with a length of 10 pixels.

2.3 Result
When we timed the execution of this benchmark on various systems it yielded the following results:

- SUN 3 (68020), single user: 21 seconds
- SUN 4 (SPARC), 2 users (load average 1.7): 3 seconds
- SPARC station 1, single user: 2 seconds
- SPARC station 1+, 3 users (load average 1.0): 1 second

When we compare the time for the SUN 3 benchmark with the approximate 20 seconds needed to draw a real picture (of about 10,000 lines) with the program using FIGARO, we may conclude that our benchmark gives us a good idea of the performance which can be attained without additional hardware. It also shows that the use of FIGARO slows down things by about 30%. This seems a little more than we expected at first, but the FIGARO implementation of PHIGS is written in FORTRAN (not the best language for this kind of application) and not specially tuned to the SUN workstations. From these results we can draw some conclusions: if we use a SPARC station 1+ the simulator can draw at the required speed, but it doesn’t leave much room for overhead caused by the calculations needed to do the simulation. Nor does it allow for the driver programs and the communication between the drivers and the simulator to consume any time. So from all this we conclude that we should look for a way to accelerate or reduce the amount of drawing to be done.

2.4 Open questions
After this experiment there are still some open questions. For instance this experiment learns us nothing about the speed increase which might be attained with the use of additional hardware. Furthermore we have no data on how fast we can do things as hidden line removal and lighting/shading to improve the detail of our simulation. Also we don’t know how much time the robot and sensor drivers take, but some little tests show that the faster machines take very little time to do these kinds of simple floating point operations.
2.5 Options

A already mentioned above if we want to use PHIGS, and continue along the same lines as our existing simulation, we probably need to add hardware to attain our goal. So when we look what hardware we can use, the GX accelerator card offers up to 540,000 2D vectors per second and up to 270,000 3D vectors per second when installed in a SPARC STATION 1+[Sun1]. While these figures are of course the optimum, even under less than optimal conditions the GX accelerator card should provide the increase in performance we need. To make the best use of the card’s capabilities it is advisable to use the SunPHIGS graphics library instead of the FIGARO implementation we currently use, since it specially optimized to make use of the card [Sun2]. And even without the card SunPHIGS should be better adapted to the SunVIEW environment (a problem with FIGARO), and offer better overall performance since it is written in C instead of FORTRAN. Another option may be the development of a simulator that simplifies the picture before it is drawn to reduce to amount of drawing. When we take the existing simulation and use it to look at a robot from a distance, a lot of very short lines are drawn, which add little to the quality of the picture. And when we use a close up view a lot of the lines which are drawn are outside our field of view. This implies that when we write a simulator that can decide which amount of detail to use, i.e., which lines to draw to view a particular object at a particular distance, we might achieve the performance we want, while we will still be able to see sufficient detail close up. Of course this has as a consequence that we probably cannot use PHIGS and have to start from scratch when implementing this simulator. If we are willing to settle for a simpler visualization another option may be to make a new representation with a 3D CAD or drawing package. Since we don’t have such a package at this moment we can not experiment with this option.

2.6 New developments

At that moment a new version of the existing simulator came available and we were able to test some of the assumptions made above. This version was completely rewritten in C, and came with a new version of the PHIGS library, SunPHIGS. The person who wrote the original simulator now works on the same Esprit PLATO project at the University of Karlsruhe in Germany and they use it as the simulator for their environment [Negretto]. The simulator program now is called SIG, for Single Input Graphics. This version of the simulator runs on Sun 4 machines so that we were able to test the increase in performance on these machines. We were also able to test this simulator on a Sparc Station 1+ with GX card. When we tested the performance of the new simulator, by having it draw a robot to be simulated, it became obvious that the predictions made above were indeed true. When running on the Sparc Station 1+ with accelerator it takes the simulator less than a second to draw a picture. We cannot
time this exactly in an easy way because the UNIX time command cannot be used. The simulator takes quite some time to start up (about 10 seconds) and this stands in no proportion to the time required to draw to robot. So when we took these new developments in consideration, we decided that the best thing we could do was use the new version of the simulator. So what we now had to do was design our Simulator Environment For Flexible Assembly Systems around the SIG simulator. When we have this design we can start to adapt existing robot simulating/driving programs and implement our own programs and sensor simulators to work in SEFFAS. Furthermore since SIG doesn’t make any provisions for the integration of sensors we will have to come up with a way to do this and implement it. In the next chapter we will show how SIG works and how we can use it as the center of SEFFAS.
3 SIG — The Simulator

3.1 SIG — Introduction

SIG is based on a client-server model in which SIG acts as a server for a client program. For a program to become a client and allow it to send commands to SIG you can link it with an object file that comes with SIG. This object file contains functions which allow the client program to send commands to the SIG server. These commands can be divided into two categories: Commands which affect the way SIG shows us the objects it simulates, and commands which affect the objects in the simulation themselves. In the first category fall among others commands to set the view position, the kind of projection used and the way objects are drawn, i.e. as vector graphics or as filled polygons. In the second category fall commands to change the world model, such as the setting the position and orientation of objects and commands to alter the configuration of a object, for instance the angles and translations of the various links of a robot. When called with the right parameters these functions allow us complete control of the simulator. As already mentioned SIG stands for Single Input Graphics which means that when used this way it can only accept commands from one client program at the same time. This is a major drawback since we want to simulate multiple robots working independently, so we want to use separate driver/simulator programs to do so. To be able to have multiple programs sending commands to SIG we use a front end to SIG called MIG, for Multiple Input Graphics. MIG allows us to have several programs running at the same time, all sending their commands to SIG.

3.2 SIG’s place in SEFFAS

Based on the information available at that time we envisioned our system as follows (fig. 1):

At this time we decided to first adapt an existing robot driver/simulator ARCS
(Amsterdam Robot Controller and Simulator) to work with SIG to get an idea of the possibilities of the program. After this was done we used the framework of ARCS to create a robot/driver simulator for the Bosch TurboSCARA 800-4 robot we have in our robot laboratory. We will discuss these programs and the way they interact with SIG in another chapter.

3.3 SIG User’s Guide

When you start “mi_server” it will start “sig” automatically and “sig” in its turn will start PHIGS. PHIGS is implemented as a separate program called “phigschild”, which runs as an independent process and takes care of all drawing on the screen. After a short while you will see two windows, the lower being the View Window, in which the simulator will show us the world, and the upper window is the Control Panel which can be used to alter the way in which the simulator shows us the world. The View Window can be resized to show a smaller or greater image of the world. As you will see Phigs always uses a square view window, whether the parent window is square or not. Since we are going to use this simulator with a variable number of other programs it is not possible to come up with a default place and size suitable for all applications. But you can arrange the windows in the way most suited to your situation. The control panel (fig. 2) offers a large number of buttons and sliders which can be used to alter the view parameters. We will now give a short tour of the most important items. First there are three sliders marked Position X, Y and Z. The X and Y are the x and y coordinates of the viewpoint, i.e. the point to which we look, in millimeters. SIG starts with a perspective view and then the z coordinate of the viewpoint is 0. This point is always in the center of the view window. The Z slider determines the height from which we view the world. As you already will have noticed there is a grid drawn in the view window. This grid is 5000 millimeter by 5000 millimeter in the real world, which means that each square of the grid is 50 centimeters square. The sliders use the same scale, so when SIG starts with a default viewpoint of (2500, 2500, 1700) we are looking towards the center of the grid from a height of 170 cm. Another slider, Distance, sets the distance from the viewpoint, again in millimeters. Figure 2.
Figure 3. Three
other sliders, Angle 1, 2 and 3, determine the angles from which we look at the viewpoint in degrees. If, for instance, we want to look at the world from above we can set slider Angle 2 to 90 degrees. We can look at these angles as rotations around the three axes of the local Cartesian system of the viewpoint, as shown above (fig. 3). Slider 1 controls the rotation around the Z-axis, slider 2 the rotation around the X-axis and slider 3 the rotation around the Y-axis. Finally the slider Focus length determines the amount of perspective, just as a camera lens. Short focal lengths give a wide view with large perspective distortion, while long focal lengths zoom in on the scene, showing a smaller portion with less distortion. Above the sliders we find a number of switches which allow us to toggle a number of options. One of these options, Shading, does not work at this moment, but it will be implemented in a future version of the simulator. The Update option determines whether the simulator will redraw the picture immediately after a change to one of the view parameters has been made. When you want to change a number of parameters and you are working on a slower machine it is wise to turn Update off, make the changes and turn it on again, because if a complicated picture is to be drawn by such a machine it can take one or two seconds to generate a frame. If you 'slide' a slider all positions passed by the slider will be drawn unless you have Update turned off. You can manually update the view at any time by clicking on the Update button. The Hidden line button toggles hidden line removal. At the moment this only works when objects are drawn as solid polygons, and it slows down the simulator considerably. At the top of the window is a row of buttons which allow us to determine other, less frequently changed parameters of the simulator. All except the Quit button have a menu under them which allows further choices of which parameter to change. On the next page is an overview of this menu structure. This will be followed by a short explanation of the most important items. The user is encouraged to play with the view position parameters and the parameters under the view menu to create the view most suited for his or her needs and taste.
The Update button we already mentioned. It has two menu items Update and Redraw. The former draws the new situation, while the latter restores the picture should it become distorted by outside influences. Next is the View button which allows you to further change the view drawn by the simulator. For instance the Attributes item allows you to change the way the simulator draws an object. When you select one of these items a dialogue box appears into which you must enter the name of the object in the simulation and a number which determines the new attribute of the item. This are some example of the values you might use:

Color: 1 Green Presentation: 1 Frame Degree of detail: 1 Frame 2 Red

The item Projection allows you to choose either perspective projection (default) or parallel projection. If we change to a parallel projection the Z slider becomes the z coordinate of the view point, which now also is no longer at the center of the view window. The bottom of the view window now is 0, and the Z sliders controls how far the 0 level of the grid is above that bottom. The Build button offers us a menu with some objects we can put in our simulator's world. It was mainly meant for test purposes, since every client program can send the simulator commands to put objects in the world. Finally there is the Quit button which is pretty obvious. Remember however that clicking this button only closes the windows and halts the program. To really stop the program interrupt it with 'ctrl-C' or 'ctrl-?'.

4 ARCS — Adaptation and extension

4.1 Introduction

When we started this project we already had another robot simulator, providing both kinematic emulation and simple visualization, called ARCS, for Amsterdam Robot Controller and Simulator. This simulator was developed at our university to control and simulate the OSCAR robot. This is a 6 degrees of freedom robot designed and built by the Philips company. It kinematic structure is much like that of the well known Puma robots. It has 6 rotating links, but unlike the Puma robot OSCAR has an arm without sideways offsets. So the OSCAR can be represented as a line figure as in the following picture (fig. 4).

And this was exactly the way ARCS simulated the OSCAR robot by means of a three dimensional line model. The advantage of this type of simulation was the speed. Even when running on a Sun 3 workstation the speed was large enough to do real time simulation. That was about the only advantage because there is little to see about a line model and it only gives you a rough impression whether the simulated move is indeed what you expect. But because Unix doesn’t offer genuine real time control even this part of the simulation left something to be desired.

4.2 Adapting ARCS

So we decided to start and try to adapt ARCS to use SIG for graphics output instead of its own line figure. But there was a little snag. Because SIG offers us a detailed representation of the robot it needs an unfairly extensive description of the robot it simulates. And we didn’t have one for the OSCAR robot. Furthermore we didn’t have a 3D CAD drawing program suitable to make such a description. So for the time being we decided to use the description of the Puma 260 robot we did have (fig. 5). As already mentioned this robot resembles the OSCAR robot in its kinematic structure, but not in its dimensions. But by using this description as a stand-in we would at least be able to try out SIG with a familiar and tested robot driver/simulator. Figure 5. SIG came with a few
demonstration client programs, but these programs only show some robots without moving them. Adapting ARCS so as to send commands to SIG to place a Puma robot in the simulated world at start up wasn’t a problem. Now it was time to find out how a client program could move the robots it had simulated by SIG. From the sources of SIG we learned that every object in the simulation that could be moved had a number of control values. In case of the Puma robot these control values were the 6 joint angles. So by sending the simulator a command to change these control values and then having it redraw the view we were able to move the robot. What we needed to do next was find a point in the existing simulation part of ARCS where these values were known. Fortunately this was easy as these values were passed as parameters to the procedure in ARCS responsible for the simulation. So we could simply put the command to send these values to SIG in that procedure and watch SIG draw the robot. At this time we kept the old simulation so we could compare it to the new one. When we tested this new version of ARCS it performed exactly as we expected. Except for the differences in dimensions between the OSCAR and PUMA robots the movements of the robots in the old and new simulations were the same. Furthermore the speed of the SIG simulation was close to that of the old ARCS run on a Sun 3 workstation. This proved that we indeed could use SIG as a replacement of our existing simulation.

4.3 Adding a user interface

While SIG offered us a nice user interface in the form of the control panel, at time ARCS had nothing more to offer than a simple command line, reading its commands from stdin. While this made it easy to execute ‘programs’ by reading commands from file rather than from the keyboard, it wasn’t very user friendly and didn’t look very professional. So as we were altering the program anyway we might as well try to add a more professional and friendly user interface. For this user interface we had two candidates: First there was the SCIL interface [Kate], the result of an earlier image processing project, which is currently being used as the front end annex user interface for two image processing programs SCILAIM and SCILIMAGE. Secondly we could use a control panel created for use with the driver program of the OSCAR robot. ARCS offered us the following commands: (P)oint to point move, (C)artesian straight line move, (G)rip, (U)ngrip, (F)ile read, (S)tatus display and (Q)uit. We will have to implement an user interface with at least the same functionality. SCIL stands for Standard C Interpreter Language and this is the most important feature of this user interface. It was developed to offer a complete development environment for image processing programs. In addition to a C interpreter SCIL also offers a menu system, on top of it, which allows the user to call compiled functions linked to SCIL and a way to enter parameters for these functions by means of dialogue boxes. SCIL also includes a command expander which allows the user to call functions with their parameters from a command line. For our ARCS interface we want a
convenient way to call the functions necessary to perform the commands listed above. We don’t want to use the command expander because ARCS already offered us that kind of interface. So we placed the commands as functions in a menu. When such a menu item is selected a dialogue box appears allowing us to enter the parameters associated with the function. When used this way we notice that this is not quite the way SCIL was meant to be used. Dialogue boxes open for each function (even if it has no parameters) and unless closed clutter ourscreen. Since it is not possible to put commentary or icons in the dialogue boxes they all look alike and when a number are open this is confusing. A solution might be to put a number (or all) of the functions under the same menu item and in the same dialogue box. This however would lead to a very large and ugly dialogue box since every parameter comes on its own line. To put in Apple terms: “The look and feel” of the interface is not right for this kind of application. So we may conclude that while it is possible to use SCIL as the user interface for a robot driver/simulator it is not the most optimal solution. It would however be possible and probably advisable to use SCIL as the user interface for vision oriented sensor simulators which might be developed for SEF-FAS in the future. For instance the use of a stereo camera for object recognition and/or sensor driven robot control. At this moment we lack a programming environment suitable for multiple robots and sensors. This falls outside the scope of this project, but SCIL looks a good candidate if it is extended with the proper routines to control and communicate with the client programs. We will return to this in the final chapter. But even if we are to use SCIL for this purpose we still need a workable user interface for the immediate control of our robots.

The next option we had was to ‘borrow’ the control panel of the OSCAR driver program. So we took the relevant source file of that program and replaced the various function calls with our own. We did not encounter any difficulties except for the single step trace option built into the OSCAR driver program. This requires the driver program to return control to the control panel after every single step in a move. To implement this in our driver/simulator would require a complete rewrite of the movement routines so we left this option out. It would be possible however to implement a single step trace option in the simulator by having it wait for a keypress after each step. But since this is not intuitive and the old ARCS did not offer this option anyway, it is not implemented. After implementing this interface and trying it out we could find only one drawback: Because it offers all the control options at the same time it is a bit large. While this may not be a problem when only one is running the screen may become crammed if two or more are active at the same time. So the user is advised to iconify (close) any control panel that isn’t actively used at a given time. However it works and looks good so we decided to stay with this user interface.
4.4 ARCS User’s Manual

You can start ARCS by executing the program arcs after you have started the simulator SIG. ARCS will now prompt you to enter the mode in which it will run. Enter “m” to put ARCS in master mode and be able to move the robot with the control panel. The use of slave mode will be explained in the chapter about the World Model Manager. Next you’ll have to wait a few moments while the simulator loads the description of the Puma robot. You will see the robot appear in the simulator’s view window and finally the control panel will appear (fig. 6). You can now enter a destination by either moving the sliders or by typing the desired coordinates after the Value: prompt. Don’t forget to click the Set button to set the sliders to these values. When entering the destination’s coordinates remember that the size of the Puma robot in the simulation is about half that of the OSCAR robot. This means that when you enter the coordinates shown in the program above the Puma robot will appear to move to about (662, 0, 350). It was easier to leave it this way so that if we get data files for an OSCAR robot we don’t have to adapt ARCS again. At the moment the adaptation of ARCS is more meant as a trial to see how this combination with SIG would work out. Next you can use the Steps slider to set the number of steps in which the simulator will show the move. On slower machines without a graphics accelerator it is advisable not to use more than 20 steps. Finally you can change the desired motion type from point to point to straight line or back. The robot will try to execute the move to the position and orientation set by you when you click the Move button. If a position or orientation can not be reached the robot will stop and a message will appear. You can open and close the gripper with the Grasp and Detach buttons but at the moment there is no provision in ARCS to pick up an object even if there is one in the simulation. See the chapter about future extensions for more information. At any time you can move the robot to its home position in 10 steps by pressing the Reset button. Clicking on Print info will print information about the position and orientation of the robot. Finally ARCS can execute off-line programs by entering the filename and clicking the Execute button in the lower part of the control panel. As already mentioned in the previous paragraph the Single step trace option doesn’t work. Off-line programs consist of the commands mentioned earlier, followed by their parameters if any. For instance the following programs first prints the current position then performs a straight line move to position (1324, 0, 700) and orientation (0, 90, 0) in 20 steps and finally prints the new position:
Note that this is exactly the way you would have entered the commands in the earlier versions of ARCS. Figure 6.
5 BOSROS — BOsch Scara RObot Simulator

5.1 Introduction

So now we had a working combination of a client and server program which proved that we could implement a robot simulation this way which suited our needs. We could now start to implement a simulation for the Bosch TurboSCARA SR 800-4 robot which we have in our robot laboratory. This is a powerful industry robot which among other things is being used for research involving vision driven robot control. It would be desirable to have a simulator to use instead of the real robot when experimenting with this type of control. We already had the right graphical data for the Bosch robot, but there were two problems: First of all the data appeared to be corrupted as SIG crashed when we tried to load it. Secondly the Bosch robot came with a complete set of hardware which takes care of driving the robot. This functions as a black box which allows us to specify a goal position and then takes care of the path planning, inverse kinematics and dynamics to have the robot move to it. So in addition to trying to repair the data files we would also have to implement a black box of our own to act as the driver/simulator program. After we succeeded in repairing the files (see appendix B) we encountered another slight problem in the fact that the Bosch description came without a gripper. Instead we used the Puma gripper as this is similar to one of the grippers we have for the Bosch. But these and the problem with the OSCAR robot shows that it might be advisable to obtain a 3D CAD drawing program capable of creating files in this format or some format that can be converted to it. Together these files give us a quite workable simulation of the Bosch robot (Fig. 7).

5.2 BOSROS — The driver simulator

Now we had the files working it was time to implement a driver/simulator program. Because we wanted the same functionality as ARCS offers we decided to replace the kinematic routines in ARCS with routines suitable for the Bosch SCARA robot. There is one difference however, while ARCS was intended to study the dynamic behavior of the robot, this is not one of the main concerns with the Bosch robot. One of the Bosch robot’s main applications is the combination of robot control and vision, so positioning is more important than speed. And since the robot can move faster than the simulator can draw anyway we decided to ignore the dynamics and just simulate the movement. It would be possible however to extend the simulator to include dynamics. Figure 7 ARCS
uses matrix calculations to solve the kinematics of the OSCAR robot. Instead of trying to adapt these routines it was simpler to use some simple trigonometry to solve the kinematics of the SCARA robot. The robot has only 4 degrees of freedom as opposed to the 6 of the OSCAR robot (fig. 8). The orientation of is controlled by one rotation and in addition movement in the Z direction depends on only one translation in that direction. This means that the X, Y positioning is controlled by two rotations. This implies that we only have to solve the 2 dimensional oblique triangle formed by the links of the robot and the imaginary line from the first rotation axis of the robot link to the last rotation and translation axis (fig. 9).

The calculation goes as follows:

\[ X = x \text{ coordinate of destination} \quad Y = y \text{ coordinate of destination} \quad A = \text{ length of first link} \]

\[ s = \frac{1}{2} (A + B + C) \]

\[ s^2 = A^2 + B^2 + C^2 - 2 \cdot A \cdot B \cdot \cos \alpha - 2 \cdot A \cdot C \cdot \cos \beta - 2 \cdot B \cdot C \cdot \cos \gamma \]

\[ \alpha = \arcsin \left( \frac{X}{C} \right) + \arcsin \left( \frac{h}{A} \right) \]

\[ \beta = - \arcsin \left( \frac{h}{A} \right) - \arcsin \left( \frac{h}{B} \right) \]

There are other ways to perform the same calculation but they don’t hold any significant advantage over this one. After the angles are calculated this way they are corrected for the configuration of the arm (left or right bend) and the quadrant in which the destination lies. Next we implement the two ways of movement: Point to Point and Straight Line. Since the Bosch controller is implemented as a black box we have no access to the functions used to perform these movements. Experimentally determining them would fall outside the scope of this project, so we decided to implement our own functions. Point to point movement is achieved by interpolating the starting and destination joint angles with a cubic spline, while straight line movement is achieved by interpolation with a cubic spline along a straight line. The number of steps in which the move takes place has no relation with any real world time. It is simply the degree of accuracy with which the movement is simulated. If we were to have a faster machine we could extend the simulation to do real time simulation. Figure 8, Figure 9.
5.3 BOSROS User’s Manual

Since BOSROS uses the same control panel as ARCS, see paragraph 5.4 for a description of this. There are some small differences between the way BOSROS functions and the way ARCS functions. First as the Bosch SCARA robot has only 4 degrees of freedom, only the first four sliders are used (fig. 10). Setting the last two sliders has no effect whatsoever. Furthermore since we use the right files for the Bosch robot the grid of the simulator, which is equal to 00 mm by 500 mm, can be used to determine the destination coordinates. Be careful however when setting the Z coordinate. The simulator uses a Puma type gripper and BOSROS uses the length of this gripper to calculate the translation parameter. Our own Bosch robot is frequently equipped with a much longer gripper and this will lead to a difference between the real and the simulated view. Again the remedy would be to create the necessary data files and adapt BOSROS. Finally in contrast to ARCS, BOSROS will refuse to start to execute a move of which the destination can not be reached. This is true for both point to point and straight line moves. This is done on purpose since if a destination can not be reached the command to go there was probably a mistake and to prevent damage to objects which may be in the way of the robot the robot refuses to move, giving the human controller the chance to check the entered coordinates. Note that both the simulator and the real Bosch robot don’t know about the floor or tab e and will happily try to press the gripper right through it if you tell them to do so. Figure 10.
6 World Model Manager — Design

6.1 Introduction

When we want to do a faithful simulation of a robot workstation, we need more than just a graphical simulation of the robot and its surroundings. For instance, we also want to be able to detect collisions between the robot and the other objects in the simulation. And if we have more robots working together, we also want a mechanism to coordinate the actions of the robots. Finally, we want to be able to expand our simulation with sensor simulation. So what we really want is a simulation of the entire world around our robots. The relevant data of this simulated world must be kept somewhere and be made available to whomever needs it. To do this we need a so-called world manager. In the next paragraphs, we will look at what would be the best design for such a world manager in our specific environment. In our environment, we use our own programs as clients of SIG, which include ARCS (Amsterdam Robot Controller & Simulator) for an OSCAR (Puma260 like) robot, and BOSROS (BOsch Scara RObot Simulator). These clients act as robot controllers: The user can enter commands for the robot, and these programs will try to perform these commands. The results of these programs are new states of the robots, and these are shown by the simulator server program, SIG (fig. 11). When we want to include sensors and want our robots to act on information from these sensors, we need away for the robot drivers to obtain this sensor information and for the sensors to obtain information about the world.

6.2 Our current system

If we want to implement a world manager, it will have to work closely with the simulator, so we had to take a closer look at our MIG/SIG simulator to get an idea how it is organized exactly. It became apparent to us that the way we envisioned our system wasn’t completely right. The names of the programs MIG and S G, for Multiple and Single Input Graphics, are confusing, and some might say badly chosen. Instead of one MIG accepting input from multiple clients, each client program has its own MIG. These MIGs all communicate with SIG. So each MIG has only a single input, while SIG receives input from multiple MIGs. Figure 11. At the moment a fully operational simulation system with N
6.3 The current model

It is obvious that since we are simulating robots, we already have a part of our world model in the model of these robots and their surroundings. This model is stored in layer 3 and used by layer 4 of our simulation system. It consists of groups of polygons to describe the geometry of the various parts of a robot, and of transformations to describe the relations between the parts. This data is normally kept in files, and read in at the start of the simulation. The client programs have knowledge of the robot they are supposedly controlling, but this is incomplete and unstructured data. And although there is communication possible between the various layers in our current environment, at the moment there is no provision for use of data in the lower layers by the clients. So what we have are a collection of polygons and the relation between various groups of these polygons, which are not readily accessible.

6.4 Desires, wishes and needs

What we want is a world simulation that contains all data normally used by a robot workstation regarding its surroundings and itself. In case of sensor driven controllers this means that we want to be able to have our controllers obtain data from the sensors in our world. These sensors will probably be simulated by separate processes, which in their turn also need information from the world (since this is exactly their job). So what we need is a world model that can be accessed by both controller and sensor simulators.

6.5 Solutions

Some of the data we need is already present in our simulation, i.e. the geometry of the objects in our simulated world together with their position and configuration.
The problem is that we cannot easily access it. To be able to do so would require a major extension of SIG. Furthermore it would drastically increase the information stream between layer 1 and layer 3. Since this data has to be present in the SIG layer we have two options left: Either we implement the World Model Manager as a separate process which keeps a copy of that data, or we extend SIG to include the World Model Manager. If we implement our World Model Manager along the same lines as the graphic simulator this will lead to the following situation (fig. 13):

![Diagram](Diagram.png)
we choose the second option there will be little change in our system, since there will be no new processes or communication lines (fig. 14):

![Diagram](image)

### 6.6 Conclusions

The first option is certainly the most flexible, it allows us complete freedom in the way we want to implement our world model and we are able to run the World Model Manager on a different workstation than the one(s) we use to run the clients and the graphic simulator. Furthermore it prevents us from having to change SIG. But there are a number of substantial drawbacks to this approach. First of all it will increase the number of separate processes from $2N+3$ to $3N+5$. This will increase the memory, processor and communication overhead considerably. Secondly since we’re using a large data structure we introduce more memory and processor overhead in maintaining it. Thirdly the whole system will become rather unwieldy to run and maintain for the user. If we expand the existing SIG program to include our World Model Manager, we do not introduce any new processes, and the communication overhead will be less because we do not need to tell our world model of changes like new objects, positions of objects and control values of objects, since these will be send to SIG any way and we can intercept them, or use the data kept by SIG. As I’ve already mentioned we do not need to keep a separate copy of our object data, since we can use SIG’s. Furthermore we do not introduce any new difficulties for the user. He can use the new system the same way he would use the old one. With the expectation that future users of this system won’t have access to or will want to use multiple workstations at the same time, the drawback that the World Model Manager runs on the same machine as the graphic simulator and can’t be run on a different one is not a problem. So if we consider all these arguments it seems that the best...
thing we can do is implement an extension to SIG to become our World Model Manager. Graphically this SIG-WMM combination will look as follows (fig. 15): In the next chapter we will show how the World Management and the extended communications will be implemented. We will implement an example in which we have a slave robot following a master robot, guided by information from a force sensor held by both robots.
7  WMM — Implementation

7.1  Introduction

This chapter will go into the technical details of the complete simulation environment and can serve as a guide when one wants to implement his own robot or sensor driver/simulators for use with the simulator. First we will explain how the design from the previous chapter for a world model manager is implemented and later we will show how it can be extended to be used with other robots and sensors. What we are going to implement is a way to exchange information between our world model and the client programs. The information flow goes in both directions: The robot for instance send information about its position to the world model, while a sensor retrieves this information and uses it to calculate a sensor value. This sensor value is then passed back to the world model so that it is available for whomever is interested in it. SIG as it is offers us a way to send information to it. We simply use the same mechanism that is used to send commands to it, to send information to our world model manager. We can then intercept these messages in SIG and pass them on to our own routines. There is however a problem with the communication in the other direction. This becomes apparent when we look at the way the communication is implemented in SIG.

7.2  SEFFAS’s communication scheme

The SIG part of the simulator is implemented as a program which reads its commands from ‘stdin’, which on the Sun workstations under Unix normally means the keyboard. So when we would start the program called “sig” we could type in commands on the keyboard and SIG would try to execute them. This is of course not what we would like. We want our robot driver/simulator programs to give the commands. This can be accomplished by using a pipe to connect the ‘stdout’ of our client program to the ‘stdin’of SIG. However this prevents us from using more than one client program at the same time. So instead of using just SIG we use the front-end “mig_server”. When we start this program it starts SIG and connects its ‘stdout’ to ‘stdin’ of SIG and vice versa. When we now start a client program, this program makes contact with the mig-server be means of a communication construct called a socket. This socket allows two way communication between the two programs. Next mig-server forks off a copy of itself (starts a duplicate of itself) which then waits for another client program. If another client is started and makes contact with this duplicate the whole process repeats itself. All copies of mig-server will use the same pipes to SIG. To prevent multiple mig-servers from using the pipes at the same time read/write semaphore (a construct used to prevent multiple clients from using the same resource at the same time) on the pipes. This leads to the following communications protocol: When a client wants to send a command to SIG it first sends it to its mig-sever
which then tries to set the semaphore to get read/write permission on the pipes. It keeps trying until it gets permission and then writes the command it received from the client to the pipe to SIG. What happens next depends on whether the client program has set answer mode or not. If it has, it will wait until it receives an answer from SIG whether the command was successfully executed. So if answer mode is set the mig-server waits for an answer from SIG and then passes it on to its client. It then releases its read/write permission so that other clients can send their commands. While this protocol and its implementation functions well when giving commands and receiving immediate answers it makes it impossible for SIG to send information, for instance about changes in the world model, to the client programs. If we want full two-way communication we will have to rewrite the entire communications routines of all the programs involved: sig, mig-server and the client programs. As this was outside the scope of this project we decided not to change the communication routines but to slightly alter the protocol to allow the client programs to poll the world model manager for changes in the world model which would affect them. We implemented a routine which intercepts messages meant for WMM and passes them on to a routine which determines which data item or items we are interested in and then calls the appropriate routine to retrieve the value of that item. This is because, as we are using independently running programs, we cannot guarantee that every program takes its turn in a regular sequence and if we do not implement such a mechanism client programs will miss certain changes in the world model.

7.3 Implementation aspects

We will have to confirm to the communication method offered by SIG which is based on strings. With these strings we will have to implement three kind of messages: First the initial report of a client to WMM, secondly the request/inquiry send by the client, and thirdly the transmission of data either as reaction on a request/inquiry (by WMM) or as world state information (by a client). We decided on the following structure: The string starts with “WMM:” to allow our routine in IG to recognize and intercept our command. This is followed by a client number, under which the client is known to WMM. In case of a new client this client number is 0. The rest of the string is determined by which kind of message is to be send. In case of the initial message the 0 is followed by the client ID. Every client program in SEFFAS should have its own unique ID. To be able to implement a data dependency mechanism a client should know on which other client programs it depends for information. The client ID’s of these programs follow the client’s own ID in the initial message. So in case of a sensor (ID=1) depending on the actions of one robot (ID=2) the initial string to be send may look as follows:
For easy construction of this string a number of constants are defined in the file "wmm.h" (see appendix D). The ID of every client to work with SEFFAS should be defined in this file. The construction of the string can now be done with the following statement:

```c
sprintf(string, "WMM: %i %i %i\n", NEW_CLIENT, SENSOR_ID, ROBOT_ID);
```

This string can then be send to SIG and the answer received in the following way:

```c
mig_send(string); mig_receive(answer);
```

The answer which the client receives is also a string which in this case only contains an integer. This is the client number under which WMM knows the client and should be used for further communication with WMM. When a client changes something in the world model it will have to notify WMM. To do this a client can send a string containing the command to do so together with the new data. For instance if we want to change the 4 joint parameters of a Bosch robot we can build a string as follows:

```c
sprintf(string, "WMM: %i %i %lf %lf %lf %lf\n", client_number, SET_BOSCH_INFO, joint1, joint2, joint3, joint4);
```

`SET_BOSCH_INFO` is a command number which should be defined in the "wmm." file. Every command needs its own routine in WMM. So if we want to keep track of the joint parameters of the Bosch robot we will have to write a routine which reads the 4 values from the string and stores them somewhere safe. In case of simple data items the routine necessary to retrieve this information can be implemented in the file "wmm.c" (see appendix E). To allow the routine to be called when WMM receives the command above the command number should be a case in the switch statement in the wmm_h ndle routine in this file. Complex data item handling routines can be implemented in separate modules, as long as they are called as a result of the switch statement in wmm_handle. Because of the great diversity in data which can be available in a (simulated) robot cell it is impossible to give standard data structures for these items. Anyone who wants to extend WMM will have to provide his own data structures and the routines to put them in a string. At the moment SIG limits the string length to 1000 characters so if very large data structures are to be passed it might be advisable to use files as an intermediary and only transmit file names via the strings. Now another client can request information about the robot from the world model manager by sending a string with its client number and a command number. Again we will have to implement a routine in WMM which in reaction to the command retrieves the data stored and puts in a string which can then be send as an answer to the client. For an example of how all this exactly works look at the files "wmm.c" and "fsensor.c" (appendices E and F). To solve the problem
of clients changing the data before other clients have taken note of it, we have implemented a semaphore-like mechanism. WMM keeps a two-dimensional array of dependencies based on the information provided by the clients when they make the initial contact. It also keeps an array of flags which indicate whether a client has noticed the change or not. A client is not allowed to change a data item before all clients which depend on it have taken note of the old value. If it tries to do so, it will receive an answer string containing a “WAIT” value, indicating that the item has not been changed and that it will have to try again. If all clients have been notified, WMM allows the data item to be changed and returns a “OK” value to the client. As a simple example, consider one robot with a force sensor in the gripper. If the robot (simulator) closes the gripper, it is not allowed to open it until the sensor (simulator) has taken note that the gripper has been closed. The diagram on the following page (fig. 16) shows us the control flow in
WMM.

7.4 SEFFAS in practice

We already mentioned the example of the two robots linked by a forcesensor and now we have implemented our World Model Manager it is time to put it all together: In this example SEFFAS will consist of the following elements: SIG, WMM, 2 BOSROS’s and FSENSOR. Which means that we will use two Bosch robots for our example coupled by the force sensor FSENSOR. Before we make some changes to BOSROS to allow us to run it in ‘slave mode’ and add the appropriate routines to WMM, we must implement our force sensor. Implementing a faithful force sensor would mean that in addition to the direction of movement we would have to take into account the mass of the robots, the acceleration of the master robot, dynamics (compliance) of the force sensor, de-
lay between the master’s movement and the slave’s reaction, and perhaps more. But none of these is available in the current simulation or even in our real world environment. Again obtaining this data and implementing it in the simulation would be a project in itself so we decided to simplify our problem. When we look at BOSROS we see that our unit of time is defined in terms of the number of steps in which to execute a move. The larger the number of steps the longer a move will take. But also the smaller the change in position in one step. If we set the delay time to one step, add to this a sensor which is compliant enough to allow for the movement to take place and discard all other factors we can base our simulation on the fact that the force on the sensor will be linear with the size of the movement step and in the direction of the movement. So when we take the difference between the old position and the new position of the master robot we have a vector which gives us the size and the direction of the force on the sensor. This is true for movement in the X, Y and Z direction of the robot but our robot has an additional rotation which changes the orientation of the gripper. If we imagine our sensor as a stick held in the grippers of the robots changing the orientation will also lead to a force on the sensor (fig. 17). To implement this force we take the end of the stick (in the gripper of the slave robot) as the action point for our force and we also take the difference in orientation as a fourth component of our force. So now we have a sensor which takes the difference between the old position and the new position of the end of the imaginary sensor together with the difference in orientation as a measure of the moment applied to it. To put it in pseudo code:

\[
\begin{align*}
X_{\text{Value}} &= (\text{New}_X - \sin (\text{New}_O) \times \text{Sensor Length}) - (\text{Old}_X - \sin (\text{Old}_O) \times \text{Sensor Length}) \\
Y_{\text{Value}} &= (\text{New}_Y - \cos (\text{New}_O) \times \text{Sensor Length}) - (\text{Old}_Y - \cos (\text{Old}_O) \times \text{Sensor Length}) \\
Z_{\text{Value}} &= \text{New}_Z - \text{Old}_Z
\end{align*}
\]

We could put together a function to more faithfully represent this force but when we remember that we will only use it to have the slave robot ‘follow the leader’, we realize that we can use it directly to have the slave robot to a new position, resulting in a negation of the force placed on the sensor. So our master – sensor – slave interaction will go as follows: The master does a movement step and sends its new position to the world model manager. The sensor which polls WMM for

![Diagram showing sensor in old and new positions with vectors representing changes in position and orientation.](image-url)
a change in this position receives the new position, calculates the difference as shown above and sends this as a new sensor value to WMM. The slave which in its turn polls WMM for the sensor value sees the new value and uses it to move to a new position. This means that in this particular case our world model will consist of nothing more than the position of the robot and the sensor value which is a result of the movement of that robot. So now we put the necessary functions in WMM and BOSROS. We extend BOSROS with the option to put it in ‘slave mode’, which means that it will not take its input from a control panel but instead starts polling WMM for the sensor value. When there is a force registered by the sensor the slave robot moves to the position indicated by the sensor value. We also add some code which sends the new position of the robot to WMM when BOSROS runs in master mode. A less elegant way would have been to add code to WMM to calculate the new position of the robot based on the new link control values (the angles and translation). This would have saved us the additional sending of the position. Our FSSENSOR program simply consists of a loop which polls WMM for a change in position of the master robot and when it receives one sends the new sensor value to WMM and shows it graphically in a window. So now we are ready to put this example to work. First we start the SIG-WMM combination as we used to, by starting migtserver. Next we start two BOSROSes. We can first start a BOSROS in slave mode and put it in the background and then start a BOSROS in master mode in the foreground or start them in separate windows. It then doesn’t matter which we start first. We must now align the two robots so we can place our imaginary sensor between them. As we don’t have control of the slave robot and our sensor has a fixed length we will have to move the master to (800, -100, 193, 0) using the control panel. This will place the master directly in front of the slave, which is still at its home position. And we can now put our sensor between them by starting the program fsensor. From now on the slave will try to follow the master (fig. 18). Be careful not to force the slave to go somewhere it can’t because the simulation doesn’t incorporate exception handling and the slave will get hopelessly confused. Figure 18.
8 Conclusions and recommendations

8.1 Conclusions

When we look back at the goals we had set, namely a simulator environment capable of simulating multiple robots and sensors and suitable for interactive use, we can draw the following conclusions: Although our experimental set-up of two robots and sensor has little practical use, it shows us that we are able to use SEFFAS as a simulation environment for multiple robots and sensors. We can easily extend SEFFAS for use with other robot drivers and sensor simulators. Since on a standard Sun SPARC workstation it takes about a second to perform one step in the simulation which includes two robots, we can use it interactively without unacceptable delays between each step. SEFFAS also shows us that with this simulation we are operating at the limit of the processing power of simple workstations. Especially if we are using multiple robots and sensors the whole simulation will down depending not only on the complexity of the robots to be visualized but also on the sensor complexity, i.e. vision based sensors like a camera will require a lot of processing time. The use of a workstation with a graphics accelerator removes the limitation imposed by the drawing of the picture, but it introduces a new one in the form of the communication. The communication scheme used, sending strings over `stdio' channels, is certainly not the best possible. According to the implementor of SIG it takes about 30 ms to transmit a string. Thus the use of polling by the clients wastes computing power. While workstations like the Sun SPARC's we are using offer the possibility to provide better and more detailed pictures of the simulated cell, without additional hardware they lack the power to do so interactively. The lack of a geometric modelling package limits the extension possibilities of SEFFAS. If we want other objects or robots in our simulation we will need the geometric data in order for SIG to be able to show them.

8.2 Future developments

As already mentioned the University of Karlsruhe works on the same project. The SIG simulator is not completely finished so when a new version becomes available it is advisable to update our simulator. Furthermore it is advisable to keep an eye on further developments from Karlsruhe especially the way they plan to incorporate sensors in their simulation although the progress of the project is not known. When we look at our own future there are a number of points of the simulator environment which can be improved. First there is the addition of a programming environment suitable for programming multiple robots. This environment must be able to access sensor information and communicate with multiple programs at the same time. If we are to use SCIL as a programming environment for our robot, SCIL would allow us to write our programs in C and this is a major advantage. But we don't want to program one robot, we want to be able to pro-
gram a complete robot cell with multiple robots and sensors. All these robots and sensors are independent running programs and SCIL only allows us to a cess functions of the module(s) linked to it. So in this form it can not be used to control a complete cell. To be able to do this we will need to add our own communication and control routine to SCIL. Another point of irritation which may stand in the way of future developments is the limited communication scheme. We would like another communication scheme which allows us to directly exchange information between the programs. One way to implement this would be message passing. In addition to this there is the general movement to a standardized windows environment for UNIX machines called X-Windows. Many programs developed by our university at this moment already use X-Windows. If we want to port SEFFAS to X-Windows we will have to use another PHIGS library for SIG and we will have to reimplement our control panels, but in addition to conforming to a standard, using X will offer us a new way to implement our communication. X-Windows supports message passing and we can use this to implement two way communication between all the separate programs. It would also make the msg_server program obsolete. When we look at the future of our robot and sensor simulations there will be the need to implement more realistic simulators. We might look at object oriented programming to implement these simulators, which would also nicely complement the object oriented approach of the X-Windows environment. A interesting project would be to implement a force based sensor in the gripper of a robot which would be able to determine the object's design, where the raw sensor data is available in our world model, [Weller] suggests the use of so-called sensor primitives to create abstract data types which cover all types of sensors. In case of more complicated sensors like cameras not the image data but the result of interpreting the data is available. It may be interesting to examine if and how these sensor primitives may be used in SEFFAS.
A Installation

To be able to run the simulator program SIG and its clients a number of environment variables has to be set. Here follows a list of these variables their current value and their function:

**MIG_BIN** /usr/koninga/DiSiSy/bin44
This is the directory in which the “mig_server” executable is kept.

**SIG_BIN** /usr/koninga/DiSiSy/bin44
This is the directory in which the “sig” executable is kept.

**PHIGSDIR** /carol/usr/lib/phigs1.1/lib
This is the directory in which the PHIGS library files are kept.

**DISISY** /usr/koninga/DiSiSyMIG /usr/koninga/DiSiSy
This is the directory in which the subdirectories “OBJECTS” and “GEOMETRY” are. These two directories contain the data files used by the simulator. In this “DiSiSy” directory are also all other directories related to the SIG simulator package.

**MIG_HOST** ‘hostname’
This variable has to contain the name of the machine on which the “mig_server” program was started and must be defined on the machine on which the client programs are to be run. This is true even if both clients and simulator run on the same machine. It is advisable to use the predefined variable “hostname” to set “MIG_HOST” on log-in.
Finally the source and executables of the client programs, like ARCS and BOSROS are in the directory “/usr/koninga/arcs”.

36
B Graphical data problems

When implementing the Bosch simulation we encountered a problem with the data files used by the SIG simulator. When we examined the data files we not only found that one of the files contained two copies of the same data, but also that the files were of a slightly different format. These files consist of ASCII data describing the polygons in a picture as follows:

```
POLYGON { 4}  POLYGON 4 -400,-400,0 N 0,0,-1 -400,-400,0 N 0,0,-1 -400,-350,0 N 0,0,-1 -400,-350,0 N 0,0,-1 -350,-350,0 N 0,0,-1 -350,-350,0 N 0,0,-1 -350,-400,0 N 0,0,-1 -350,-400,0 N 0,0,-1
```

The remedy is obvious: Place the curly brackets around the number of vertices in the polygon. The cause of this problem probably lies in the version of the program used to create the files.
C Changes to SIG

The following changes have been made to the source files of the SIGsimulator package:

- `sig.c`: Added following line to fix a bug on line 4279: `sig_set_light_mode(0);`
- `sig_interface.c`: Added the following to intercept WMM messages and pass them to the protocol:
  ```c
  #if 0
  skip_rest_of_line(&command_bp);
  #endif
  ```
- `makefile`: Changed the various directories to suit our needs and added "wmm.c" and "wmm.o" to the files to be compiled and linked to "sig"
file: "wmm.h"-----------------------------------------------/* This are the commands on which WMM reacts */
#define SET_SENSOR_VAL 3
#define GET_SENSOR_VAL 4
#define SET_BOSCH_POS 5
#define GET_BOSCH_POS 6
#define SET_PSENSOR_VAL 7
#define GET_PSENSOR_VAL 8
/* This are answers from WMM */
#define WAIT 0
#define OK 1

/* This are client id's, every client should have */
/* and master clients of the same name. SLAVE_ID should */
/* be greater than the heighest client id. */
#define MASTER 0
#define SLAVE 256
file: "wmm.c"

#include <stdio.h>
#include <stdlib.h>

static int notified[MAXCLIENTS+1]; /* Array with notified flags */
static POS pos = { 0.0, 0.0, 0.0, 0.0 }; /* Initial unknown robot position */
static POS val = { 0.0, 0.0, 0.0, 0.0 }; /* Initial unknown sensor values */

int client_slot(id) int id;

int clear_notified(client) int client;

int all_notified(client) int client;

int get_int(cbpp, &client); int client;

int all_notified(client) int client;

int get_double(cbpp, &pos.x); int client;

int get_double(cbpp, &pos.y); int client;

int get_double(cbpp, &pos.z); int client;

int get_double(cbpp, &pos.o); int client;

int get_sensor_val(client) int client;

int get_bosch_pos(client) int client;

int set_bosch_pos(cbpp, client) char **cbpp; int client;

int set_psensor_val(cbpp, client) char **cbpp; int client;

int get_psensor_val(client) int client;

int new_client(cbpp);
```c
#include <suntool/sunview.h>
#include <suntool/canvas.h>

#define TRUE 1
#define FALSE 0
#define MAX(a,b) ((a)>(b)?(a):(b))

int npos[4] = {0,0,0,0};
POS nval = {0.0, 0.0, 0.0, 0.0};
POS oval = {0.0, 0.0, 0.0, 0.0};
int first = TRUE;

void draw_bar(int x1, int y1, int x2, int y2)
{
    pw_vector(Pw, x1, y1, x2, y1, PIX_SRC, 1);
    pw_vector(Pw, x2, y1, x2, y2, PIX_SRC, 0);
    pw_vector(Pw, x2, y2, x1, y2, PIX_SRC, 0);
    pw_vector(Pw, x1, y2, x1, y1, PIX_SRC, 0);
}

my_sensor_func()
{
    if (oval.x>0.0) draw_bar(150, 70, 150+MIN((int)oval.x, 100), 90);
    else undraw_bar(150+MAX((int)oval.x, -100), 70, 150, 90);
    if (oval.y>0.0) draw_bar(150, 120, 150+MIN((int)oval.y, 100), 140);
    else undraw_bar(150+MAX((int)oval.y, -100), 120, 150, 140);
    if (oval.z>0.0) draw_bar(150, 170, 150+MIN((int)oval.z, 100), 190);
    else undraw_bar(150+MAX((int)oval.z, -100), 170, 150, 190);
}
```

F "FSENSOR.C"

file: "fsensor.c"

```c
#include <suntool/sunview.h>
#include <suntool/canvas.h>

#define TRUE 1
#define FALSE 0
#define MAX(a,b) ((a)>(b)?(a):(b))

int npos[4] = {0,0,0,0};
POS nval = {0.0, 0.0, 0.0, 0.0};
POS oval = {0.0, 0.0, 0.0, 0.0};
int first = TRUE;

void draw_bar(int x1, int y1, int x2, int y2)
{
    pw_vector(Pw, x1, y1, x2, y1, PIX_SRC, 1);
    pw_vector(Pw, x2, y1, x2, y2, PIX_SRC, 0);
    pw_vector(Pw, x2, y2, x1, y2, PIX_SRC, 0);
    pw_vector(Pw, x1, y2, x1, y1, PIX_SRC, 0);
}
```

my_sensor_func()/* Function which is called on a periodical basis. * * It should be called automatically by the SunView runtime handler. */
{
    sprintf(command, "WMM: %i %i %lf %lf %lf %lf
", my_id, SET_FSENSOR_VAL, nval.x, nval.y, nval.z, nval.o);
    do { mig_send(command);
        mig_receive(answer);
        sscanf(answer, "%lf %lf %lf %lf", &npos.x, &npos.y, &npos.z, &npos.o);
        oval.x = nval.x;
        oval.y = nval.y;
        oval.z = nval.z;
        oval.o = nval.o;
        printf("Robot pos: %lf %lf %lf %lf\n", npos.x, npos.y, npos.z, npos.o);
    } while (1);
}
```
Literature

[Gini] M. Gini
The future of robot programming
Robotica, 5, 235-246
1987

[Dillmann] R. Dillmann, B. Hornung, M. Huck
Interactive programming of robots using textual programming
and simulation techniques
Proceedings of 16th ISIR conference
1986 - Brussels, Belgium

[Huck] M. Huck
PhD thesis on Robot Programming Systems
1990 - Karlsruhe, Germany

[Rembold] U. Rembold, K. Hoermann
Programming of Industrial Robots; Today and in the future
Proceedings of the NATO international Advanced Research Workshop on
Languages for sensor based Control in Robotics
NATO-ASI Series F29
September 1986 - Italy

Off-line Programming of Exception Handling Strategies
Robot Control 1988
Preprints of the IFAC-Symposium
October 1988 - Karlsruhe, Germany

F. Arlabosse
Robot Control with Procedural Expert System
Proceedings of NATO Advanced Study Institute on Expert
Systems and Robotics
NATO-ASI Series F
July 1990 - Corfu, Greece

[Mul] Peter Mul
Robot programming and simulation system for robot control
featuring exception handling
December 1988 - Amsterdam, The Netherlands

[Roth] Harald Roth
Entwurf und Implementierung eines Visualisierungsmoduls für ein Robotersimulationssystem unter Verwendung eines Grafikstandards
May 1988 - Karlsruhe, Germany

[Negretto] U. Negretto, P. Mul
Task Planning and Simulation of Flexible Assembly Systems
G2-UKA-01.90/1
January 1990 - Karlsruhe, Germany

[Sun1] SunPHIGS information brochure
Sun Microsystems Inc.
1988 - Mountain View CA, USA

[Sun2] Sun SPARC station 1+ and 1+ GX information brochure
Sun Microsystems Inc.
1990 - Mountain View CA, USA

[Kate] Ton K. ten Kate, Richard van Balen, Arnold W.M. Smeulders, Frans C.A. Groen, George A. den Boer
SCILAIM: A multi-level interactive image processing environment
Pattern Recognition Letters 11 (1990) 429-441, North-Holland
1990 - Amsterdam, The Netherlands

[Dondorp] Er in Dondorp
ASSIM - De Amsterdam Sensor Simulator
June 1990 - Amsterdam, The Netherlands

Sensor based control for autonomous robots
Universiteit Van Amsterdam
Amsterdam, The Netherlands

43