AI/CS 4 Project
Model-Driven Assembled Circuit Board Inspection

Neil W. Rumney

May 29, 1990

In this project, a model-driven inspection system for circuit board components is described. The system is designed to exploit a modular approach allowing it to be extended easily for larger subsets of electronic components. The project is based on work done by Chan [1] but extends the model description language used to introduce hierarchical descriptions and some labour saving language features which make defining models easier. It is designed to inspect four types of component features - integrated circuit pins and bodies, capacitors and resistors.

Contents

1 Introduction

With the development of robots being used on the production line, it has been increasingly necessary in terms of quality control to automate the inspecting processes of items produced in this way. This results in less time being spent by human inspectors on these tedious jobs. The introduction of automatic inspection also has the benefit of being much quicker and less prone to mistakes that human inspectors might make after a long shift. One particular inspection task that would be particularly useful if done automatically is circuit board inspection. Increasingly, circuit boards are assembled by robot rather than by hand. The layout of circuit boards are becoming more densely packed and therefore more difficult to inspect. Assuming that a suitably high picture resolution can be achieved then automatic inspection is a good candidate.

The problem being considered in this project is to carry out a series of tests to check that a circuit board matches a predefined model. A lot of inspection systems that have been developed previously have worked with binary images [1]. This is not possible for circuit board inspection because the features being examined vary greatly in intensity and too much information would be lost if the work was done using a binary image.

The inspection tests to be carried out are designed to see that the circuit board has no structural defects after being assembled. The defects that the tests will be designed to find include bent or missing chip legs, missing components, misplaced components and extra components that should not be there. Simple tests can be defined that will check for the existence of objects in the image of the circuit board. The problem is to be able to reliably measure the properties of the component and compare them against those specified for the component in the model.

The project can be readily split into two sections. The first is the design of the controlling framework. This will use a model definition of a circuit board to guide the inspection process, calling the appropriate tests where needed. The second section is the design of reliable tests. Inspecting a circuit board involves the following five steps:

- Capturing an image of the circuit to be inspected.
- Creating a model definition of the circuit.
- Parsing the model definition.
- Checking the semantics of the model.
- Applying the tests referenced in the model to the image features. This requires transforming from model to image co-ordinates.

The structure of the controlling framework is based on the work of Chan [1]. In his paper he describes a method for the inspection of assemblies according to a geometric model. He developed a language for creating models using features consisting of lines, holes, corners and arcs. I have taken this as a basis for an extended language which is more suitable for describing circuit boards. The extended language allows for hierarchical object definitions that would not have appeared in Chan's assemblies.

An important consideration with the design of this project is that it should be easily extended to take account of more component features and feature tests. Therefore the framework should not depend upon some static definition of the modelling language being used.

A small example of what has been accomplished is shown below. The model below describes an IC positioned at the left hand side of the circuit board (see figure 1). The right hand row of pins is to be inspected. When this is done the window around the pins is highlighted and a copy of the window is positioned in the upper left hand corner of the image. This copy is how the test *sees* the area to be inspected. The scan line along which the pin test is done is shown in the upper left hand window.

```
CIRCUIT memory
WITH
        COMPONENT chip ic
                POSITION(15,200)
                ORIENT(-90)
ENDCIRCUIT

COMPONENT chip
WITH
        IClegs: iclegs2 WINDOW(135,15) AT(135,50,180)
                PINCOUNT(135,15,8,5)
ENDCOMPONENT

END
```

There are three other features available in the component library - ICbody, RESISTOR and CAPACITOR. These can all be used in the same way as IClegs except that different tests would be used than PINCOUNT.

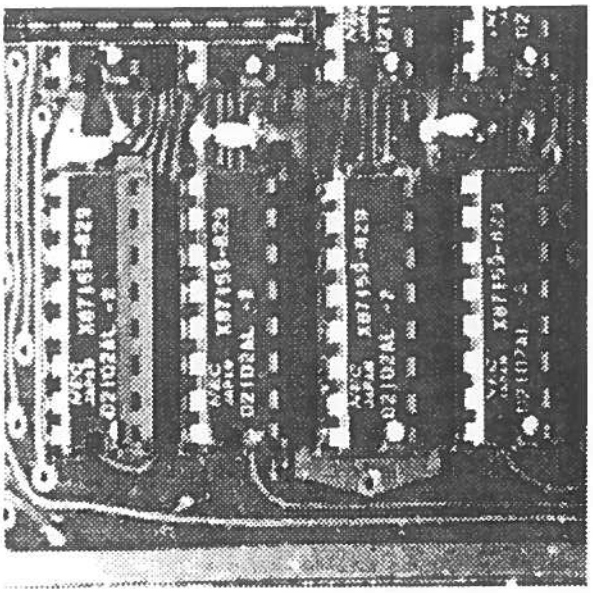

Figure 1: Display on Framestore

2 Overview of Implementation

This chapter describes the modules that constitute the inspection system. There are four modules:

- Model Data and Loader
- Inspection Control Mechanism
- Inspection Tests
- Windowing System

2.1 Model Data and Loader

The model data describes the circuits that are going to be inspected. The information given in the model descriptions gives the contents of the circuit, positional data for the components within the circuit and also information on how to identify whether particular components are correctly positioned and undamaged. The model structure and examples are discussed in more detail in chapter 3.

The model loader consists of a parser that has been written using the UNIX©tools YACC and LEX, and a model linker. The parser has been written so that all the relevant information held in the model data files is stored in a more convenient and compact form for use during the inspection process. The parser does not attempt to try any error recovery. If any syntax errors occur then the program will stop.

The output from the parser is a tree that has two main types of node. The first describes a type of component and what subcomponents it contains. The second type gives instances of component definitions. These instances contain information relating to the position and orientation of the particular components that appear in the circuit.

The inspection process requires access to both the definition of a component and also the instance of the component before it is able to inspect the components. To make these accesses easier the parse tree produced earlier is modified slightly by linking up each instance with its corresponding definition using the model linker. This is carried out after the parsing has taken place. It merely re-organises the layout of the data.

2.2 Inspection Control Mechanism

The inspection process applies tests to the specified features of the components specified. All of these components may not necessarily be referenced at the top level of the model definition. This is because components can be defined to contain subcomponents in their definition. The process requires that all the components appearing in the circuit definition be inspected.

Each component can contain instances of three forms of feature - a subcomponent, a subcomponent defined inside a repeat loop, and one of a number of predefined features. This means that one of three things must happen depending upon the type of feature in the component.

- If a non-repeating component is found then this subcomponent can then be inspected using the same process.
- A repeating component requires that the component referenced must be inspected the appropriate number of times with varying positional parameters given each time.
- A predefined feature has a list of tests which have to be applied to it. If all the tests succeed then the feature is said to have been tested correctly.

The inspection process can be viewed in the following way:

```
Select circuit model definition

repeat
        get component
        if component is in repeat loop then
                inspect component once for each position
        else
                inspect component once
until no more components
```

where *inspect* does the following:

```
repeat
        get feature
        if feature is in a repeat loop then
                inspect subcomponent once for each position
```

```
        if feature is a subcomponent then
                inspect subcomponent
        else
                apply tests to feature
until no more features
```

The recursive nature of the process can be seen in the inspection of subcomponents. Each definition of a component can have subcomponent definitions within it. It does not matter in which order the inspection is done since all the inspections have to be carried out.

2.3 Inspection Tests

The inspection process does not search over the whole image to find the features that it is inspecting. The model definition gives windows to direct the inspection of the particular features being referenced. A series of tests are then applied to this area of the image. If all succeed then it is assumed that the feature has been found and is not defective. The whole process therefore depends on the tests being sufficiently rigorous to identify the presence and correct composition of the component in question. The tests to be applied are detailed in the model definition for that circuit board. Each test will be used to find out specific things about the feature such as its orientation. The tests themselves will convey any necessary information back about the reason it might have failed.

2.4 Windowing System

The windowing system developed for the program is used when inspecting the features of each component. It converts from the model co-ordinate system to a world or screen co-ordinate system. The complication encountered here is that more than one transformation may be required. Each component defined has its own co-ordinate system as does the circuit board itself. Therefore if a particular component being inspected is the subcomponent of another then two transformations have to be carried out rather than a single one for the local co-ordinate system. A final transformation is then required to get to screen co-ordinates.

For each component there is a window with origin (x, y) and orientation θ relative to its super-component definition. At the top level we have the circuit board with position and orientation relative to the screen. At the next level down we have component window positions relative to the circuit board. The subcomponents are positioned relative to the components (super-components) that are defined in terms of the subcomponents.

The aim of the windowing system is threefold:

- Transform from component co-ordinates to screen co-ordinates
- Displaying the windows on the screen for debugging and user feedback
- Putting the window data into an appropriate form for applying tests

2.4.1 Transforming Co-ordinate Systems

Transforming from model co-ordinates to screen co-ordinates may require multiple transformations, one for each level of subcomponent nesting. For n components each of which is a subcomponent of the next we have $n+1$ transformations to get from component n's co-ordinate system to screen co-ordinates. The extra transformation is required to get from circuit co-ordinates to screen co-ordinates. This process can be viewed as the following matrix multiplications combined together to give a single transformation:

Take the transformation matrix for moving from one co-ordinate system to another to be

$$\begin{pmatrix} cos\theta & -sin\theta & x \times s \\ sin\theta & cos\theta & y \times s \\ 0 & 0 & 1 \end{pmatrix}$$

for a window with origin (x, y), orientation θ and scale factor s. This implementation uses a single scale factor which converts from the model co-ordinates to screen pixels. It is therefore assumed that all the models have the same scale factor, i.e. are defined in terms of millimeters.

Let T_c to be the transformation matrix from circuit co-ordinates to screen co-ordinates. This matrix has a slightly different form from the others. A scaling factor is not used when defining the position of the circuit board to the screen origin. This is because the information is not held in the model definition and has to be given at run-time by finding out the co-ordinate of the reference point of the circuit board on the screen. Since this figure is already a screen co-ordinate then no scaling need take place. The resulting combined transformation T_{final} is given by

$$T_{final} = T_c T_1 T_2 \ldots T_n$$

where T_1 is the matrix for the outermost component definition, T_2 is the matrix for a

subcomponent of T_1 etc.

T_{final} can then be applied to the corner points of the window defined for component n giving the desired screen co-ordinates.

2.4.2 Displaying the Windows

The graphics package being used for display purposes on a framestore is a very basic one. It is not able to cope with displaying a rectangular window at all angles on the screen. It is therefore necessary to produce a copy of the screen with the appropriate changes made where the window is to be displayed and then dump the whole of this copy out to the framestore. To achieve this it is necessary to know more than just the corner points of the window. All the boundary points of the window must also be calculated.

Calculating the boundary points of a window is achieved by finding out the equations of the four lines that make up the window. From these equations it is simple enough to find all the possible x co-ordinates for each y co-ordinate of the window.

Once the boundary of the window has been calculated it is then necessary to create an image of the whole screen to be displayed. The original graphical output was developed on a framestore that could display three different pictures at once by displaying each one on a different colour plane. This was useful because the image of the circuit board could be displayed on one image plane with the windows showing where the inspection process is working could be showed on a second colour plane. This method had to be modified later to work on a different framestore that was only able to display one colour plane at any one time. Obviously there could not be two separate images used this time. It is not possible to display the window as a solid block of colour because this would blot out the part of the circuit board that is being scrutinised. The solution used was to take a copy of the circuit image and then invert the parts of the image that correspond to the window being displayed. This does not have such a desirable effect as using a different colour but it does highlight the region sufficiently.

2.4.3 Preparing the Window Area for Applying Inspection Tests

Once the outline of the window has been identified it is then necessary to get the appropriate data into a form suitable for applying tests. If the data was used directly from the screen image then the result would be that each window area tested could be at a different angle. This would mean that tests would have to be able to cope with the varying angles that the data could be given to them. It was therefore decided to rotate all the screen co-ordinates of

the windows round until they were parallel to the co-ordinate axes of the model co-ordinate system. The only information required by the tests themselves is the size of the window plus any scaling factor that might be involved.

2.4.4 Example of Windowing System

An example of the windowing system is detailed below. Figure 2 shows how two components, A and B, are related to each other and also how component A is related to the screen origin. The angles that are marked show the rotation of the component in relation to its super-component. The co-ordinates show the translation. The diagram shows that component B is positioned at point $(20, 40)$ inside component A at an angle of -90^o. Component A is in turn positioned at $(35, 5)$ with rotation 45^o to the screen's x-axis.

The co-ordinate frames for these two objects are given by the tuples

$$[35, 5, 45^o]$$

for component A relative to the screen

$$[20, 40, -90^o]$$

for component B relative to component A

Figure 1 shows the co-ordinate axes used in defining objects. It was decided to use the same axes as used with the graphical display to keep consistency. This means that the origin is the upper left hand corner of the co-ordinate system. To obtain the transformation from the co-ordinate system in component B to screen co-ordinates we have to apply the technique show earlier. Take the two matrices formed from the co-ordinate frames and multiply together to give a combined transformation. The leftmost of the matrices should belong to the outermost component definition. Thus we have

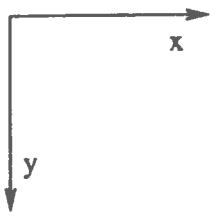

Figure 1: Co-ordinate Axes for Models and Screen

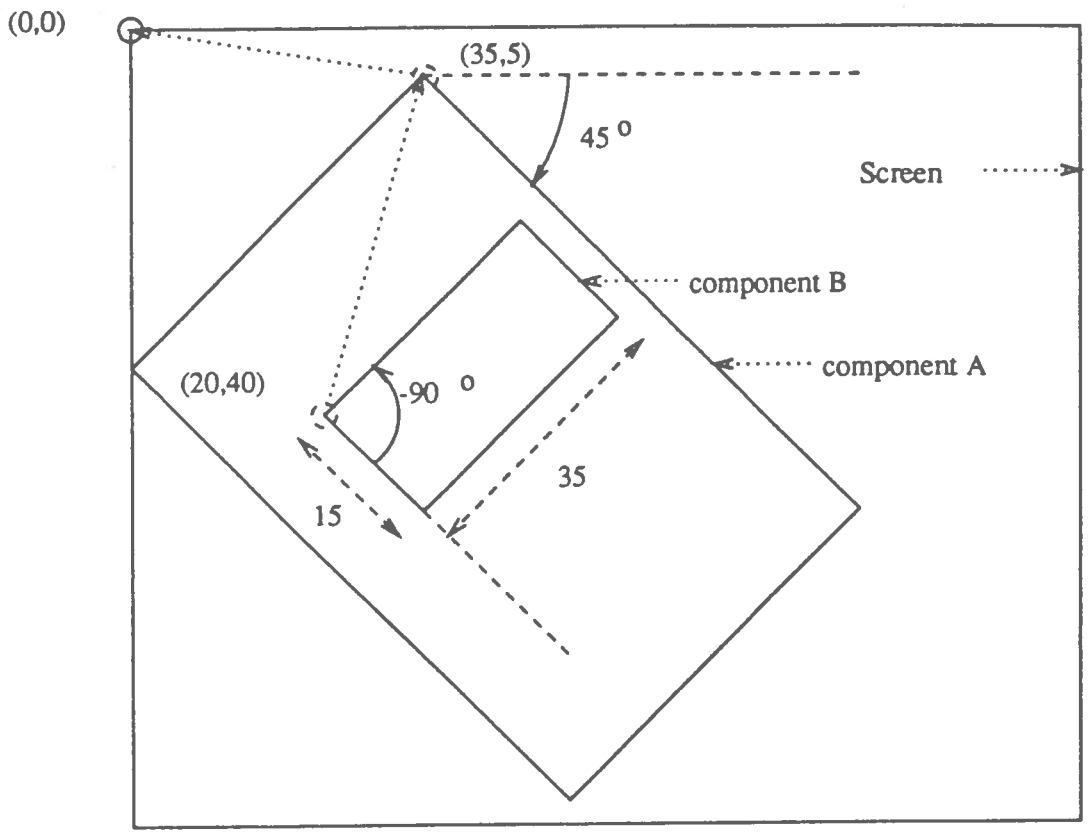

Figure 2: Pictorial View of Model

$$T_{final} = T_A * T_B$$

or in matrix form we have

$$T_{final} = \begin{pmatrix} \cos(45^o) & -\sin(45^o) & 35 \\ \sin(45^o) & \cos(45^o) & 5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(-90^o) & -\sin(-90^o) & 20 \\ \sin(-90^o) & \cos(-90^o) & 40 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0.7 & 0.7 & 20.9 \\ -0.7 & 0.7 & 47.4 \\ 0 & 0 & 1 \end{pmatrix}$$

We now have the matrix for applying to the four corner points of component B. Remember

that these corner points are defined in relation to the reference point of component B. So for length 35 and width 15, we have the corner points

(0,0) (35,0) (35,15) (0,15)

Translating to screen co-ordinates we get

$$= \begin{pmatrix} 0.7 & 0.7 & 20.9 \\ -0.7 & 0.7 & 47.4 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 35 & 35 & 0 \\ 0 & 0 & 15 & 15 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

which gives the points

(21,47) (46,23) (56,33) (32,58)

Once the screen co-ordinates have been calculated then the windowing system places the block of screen bounded by these corner points into a more suitable form for the inspection process.

3 Test Specification

A description of the circuit being tested is required so that the program is able to focus on the correct aspects of the image data. This chapter describes what kind of description is required and how the program receives it.

Section 5.1 describes a test plan that was used to evaluate the object modelling capabilities according to the specification given below.

3.1 Circuit Board Contents

The aim of this program is to obtain a reliable inspection process for different circuit boards. The inspection process is designed to examine the components that make up the circuit board. There are a large number of different type of components that can appear in a circuit board. It was decided to concentrate on only three types to demonstrate the approach:

- Integrated Circuit
- Resistor
- Capacitor

Modelling these components should allow a wide variety of circuit boards to be described. The resistor and capacitor have simple bodies. The integrated circuit is an example of a more complex component that is made up from a number of smaller parts.

Modelling a circuit board using only descriptions of these three components restricts the extent of the modelling capabilities. Other components that may be present have to be ignored. There may be tracks visible on the surface of the board which are not modelled. All of this adds up to what appears as noise to the inspection process.

What information must be modelled? To model a circuit board there are two pieces of information that must be given - a position and what to expect at that position.

The components which are to be modelled come in standard forms for each type of component that appear on circuit boards. The differences come in where they are positioned. It is necessary to have a reference point on a component by which you can define a position and orientation that it will appear on a circuit board.

Describing the component in terms of a geometric description involving lines and arcs does not give enough information. It may describe the outline of components well enough

but does not give any information about the internal appearance of the object such as how much light it reflects or what surface markings it has. A grey level image is required rather than a binary image otherwise too much information would be lost. This makes recognition of geometric features much more difficult. A line in a grey level image is much more difficult to pinpoint than it would be in a binary image.

Rather than have a geometric description of a component it was decided to use the tests themselves to describe the components. It is more of an implicit description than that given by a geometric description. The idea is to have a label for a particular feature that might belong to a component and then associate a number of tests with it. These tests are used to identify the presence of the feature. Using labels for features allows new features to be modelled by using different combinations of tests. The test can be considered as primitives that model the features.

3.2 Model Data

The data required to describe a circuit board well enough for inspection purposes has been discussed above. The position and composition of a component is required. What form should this be supplied in? The aim in defining a language to describe models is not only to get the required information but also to allow the information to be described in a concise form and without too much outlay in effort for the person describing the circuit. There are a number of points which should be considered to help with producing an easily used language:

- **Many Components:** There can be a wide diversity of components on a particular circuit board. Therefore it would be sensible to consider using a library of components for defining new circuit board models. It would save having to redefine components for new circuit boards when they have already been used previously.
- **Repeated Components:** A certain component does not necessarily appear only once on the circuit board. It would be tedious to make two or more definitions for a component just because it appears in more than one place on the board. It is much better to define it once and just reference the definition each time it is required.
- **Subcomponents:** A particular component might consist of different parts joined together. An example of this might be an integrated circuit. This has a body and pins. For such an occurrence it would be tedious to design it all as one component consisting of a definition of the body plus a number of pins. This would lead to repetition during the definition of the pins. It would be sensible to allow a component to be described in terms of features and also of other subcomponents.

Repeated components within a circuit are coped with in two ways. If there are a number of the same component which are positioned at unrelated places round the circuit then it is desirable to set up a single definition and use references to that definition within the model file. The second possibility is that a component may appear at regularly spaced intervals in columns or rows. In this case it would be sensible to define where the first of the components appear and then give the spacing between the components.

The two forms of repeated components discussed above are implemented at the top level in a model description. The top level describes what components the circuit board contains. The example below shows how the components can be described.

```
CIRCUIT <name>
WITH
        COMPONENT <definition_name> <instance_name>
                POSITION(x,y)
                ORIENT(angle)

        REPEAT <number_of_times> STEP <interval> ALONG <axis> FOR
                COMPONENT <definition_name> <instance_name>
                        POSITION(x,y)
                        ORIENT(angle)
        ENDREPEAT

ENDCIRCUIT
```

The first entry in the circuit definition is a reference to a component that must be defined elsewhere in the model file. A position and orientation is given to show where the component is to appear on the circuit board. The positioning of the component is in relation to an imaginary reference point on the circuit board. This reference point is usually taken to be at the top left point of the circuit board with a zero orientation being parallel to the x-axis. Each component has a similar reference point. This means that the position and orientation of a component is where the component's reference point will be placed in relation to the circuit's reference point.

The second entry of the circuit definition shows how to define a component that appears in regularly spaced intervals along one axis. The number of components to be positioned, the spacing between them and the axis along which they appear are all given. The reference to the component below that is used to show where the first component should appear. The others can be found by adding the interval value to the starting point. This method for defining a row of the same component is much easier than having to explicitly mention each component's position.

So far only the components that appear on the circuit board have been positioned. It is now necessary to define what these components are. A component definition is made up of features which can be in one of three forms. Two of the forms are the same as those that can appear in a circuit definition. A component can appear in the definition of another component. This is known as a subcomponent. A subcomponent can also appear inside a repeat statement.

The third form of feature that can make up a component is a predefined feature. There are four types of feature in the component library at the moment:

- ICbody
- IClegs
- RESISTOR
- CAPACITOR

These types of feature are used as labels to which groups of tests can be associated. The features themselves have no meaning to the program. A name, window and position are associated with the feature. The name is used for the purposes of user feedback. The window is defined as having a length and a width and corresponds to area of the image which will be fed to the tests during the inspection phase. The position shows where the window will be placed. It has both a co-ordinate origin and an orientation. These are relative to the component in which the feature is defined. Finally we have some number of tests which are associated with the feature. These are used to determine whether the feature is present within the window in an undamaged form. The description of the feature is effectively tied up in the definition of the tests. These tests are chosen by the person making up the model file. A feature might appear as below:

```
IClegs: iclegs1 WINDOW(135,15) AT(0,0,0)
        PINCOUNT(135,15,8,5)
```

This defines a window starting at the top left hand corner of the component. The test PINCOUNT is associated with the feature *iclegs1* and looks for eight pins with a width of at least five units. The unit length will depend upon the scaling factor for converting from model co-ordinates to screen pixels.

The overall form of a component would consist of some combination of the three features:

```
COMPONENT <definition_name>
WITH
        COMPONENT <definition_name> <instance_name>
                POSITION(x,y)
                ORIENT(angle)

        REPEAT <number_of_times> STEP <interval> ALONG <axis> FOR
                COMPONENT <definition_name> <instance_name>
                        POSITION(x,y)
                        ORIENT(angle)
        ENDREPEAT

        <feature_type>: <name> WINDOW(length,width) AT(x,y,angle)
                        test_1(parameters)

                        :

                        test_n(parameters)

ENDCOMPONENT
```

3.3 Error Recovery

The language described above has been implemented using YACC and LEX. The BNF of the grammar is shown in appendix A. At present there is no provision for error recovery. When a syntax error is encountered during the parsing of a model then the program will stop. The data being parsed is displayed during the parsing process and so when an error occurs the place where the problem appears will be shown on the screen. A small amount of semantic checking is carried out during the parsing. This involves a repeat statment where at least one component must appear, and it must be either along the x or y axis. The only other checking done is for the number of parameters given to a feature test. If the correct number is not given then the program will report this and stop.

3.4 Extending the Model

Two extensions to the modelling capabilities were considered. One simplifies the creation of models whereas the other increases the power of the system.

3.4.1 Associating Tests With Features

As it stands at the moment, in order to create a model file it is necessary for the user to know what inspection tests are required for each component that is used. It is up to the user to put the appropriate tests in. Rather than giving tests it would be better if the user could give a description of the features. These descriptions could be formed from functions relating to the features, such as SIZE(a,b) denoting the length and width of the feature. This allows the user to give a more intuitive description. The parameters of these functions could be extracted and then given as arguments to a specific batch of tests associated with the particular feature. A definition might look like the following:

```
COMPONENT chip
WITH
        ICbody: ic1 WINDOW(length,width) AT(x,y,angle)
                SIZE(a,b)

        IClegs: p1  WINDOW(length,width) AT(x,y,angle)
                NUMBER(8)

ENDCOMPONENT
```

This is not too difficult to achieve because the features, ICbody etc. are effectively labels to which tests are associated. Rather than having the user specify them for the features this extension would add the tests on before hand. The parameters of the functions describing the features would then be given to the tests.

3.4.2 Pairwise Tests

An extra set of tests could be implemented which would allow comparisons between features. These might test the separation or relative orientation between features in the image. This would require standard parameters to be passed back such as the position and orientation of the feature. These could be stored away ready for pairwise comparisons to be carried out. A definition might now look like the following:

```
COMPONENT chip
WITH
        IClegs: p1 WINDOW(135,15) AT(0,0,0)
                PINCOUNT(135,15,8,5)
```

```
        IClegs: p2 WINDOW(135,15) AT(135,50,180)
                PINCOUNT(135,15,8,5)

COMPARE
        RELORIENT(p1,p2,180,2)

ENDCOMPONENT
```

This describes a component which consists of two rows of IC pins. A pairwise test is made between these features which is used to compare the orientation of them. They should be 180° apart within a tolerance value of 2°.

Some standard tests would have to be applied to all the features regardless of what tests would normally be associated with them. These could be hidden away using the method discussed in the first extension (3.4.1).

3.5 Example Model

To show how a model file looks an example is shown in Figure 4 for the image in Figure 3. Four integrated circuits are modelled along with two resistors. The image has these components outlined to show what areas of the screen the inspection process actually deals with.

The resistors appear horizontally in the image and so the model file reflects this by positioning the parts as they are with no orientation. The chips, however, are shown rotated round by 90°. The model file also shows this (Figure 4). The four chips are grouped together into one unit. This unit is rotated through 90° so that as each chip will be correctly orientated when positioned. A repeat statement is used to define the positioning for the four chips. The first starts off at (15, 200) and there is a gap of 60 units between each. The chips are positioned along the x-axis of the component *chips*. The individual chips are defined to consist of two rows of pins, i.e. two IClegs features. One feature is rotated through 180° in relation to the other. This is so that the window given to the inspection mechanism will have the pins pointing towards the bottom of the window, away from the body of the chip.

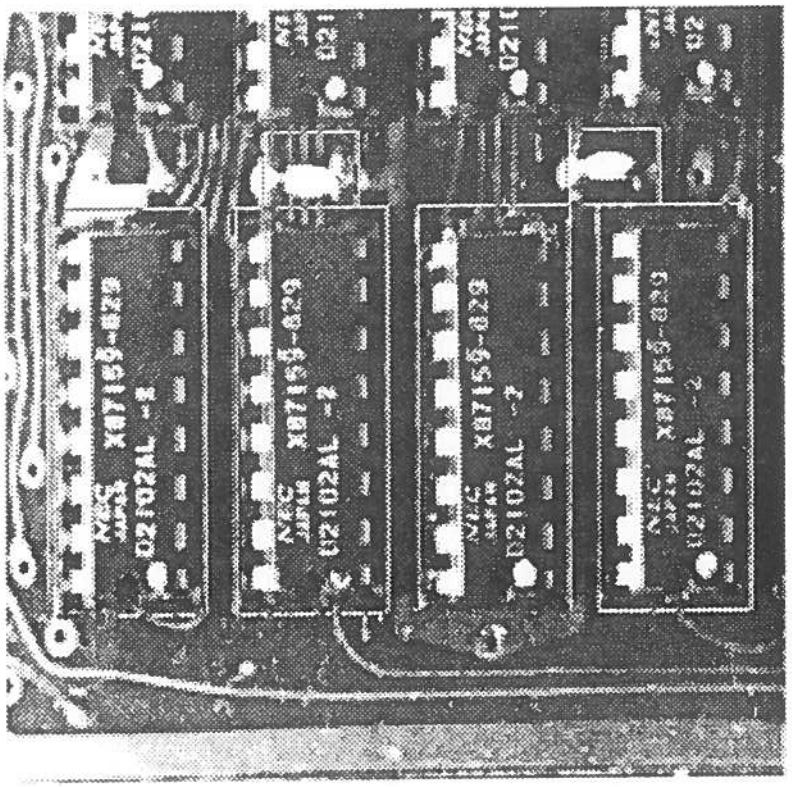

Figure 4: A Digitized Image of a Circuit Board With Components Outlined

```
CIRCUIT memory
WITH
        COMPONENT chips memchip
                POSITION(15,200)
                ORIENT(0)

        COMPONENT resistors res1
                POSITION(90,40)
                ORIENT(0)
ENDCIRCUIT

COMPONENT resistors
WITH
        RESISTOR: r1 WINDOW(30,25) AT(-5,0,0)
                LENGTH(20,5)
        RESISTOR: r2 WINDOW(30,25) AT(95,0,0)
                LENGTH(20,5)
ENDCOMPONENT


COMPONENT chips
WITH
        REPEAT 4 STEP 60 ALONG x FOR
                COMPONENT chip ic
                        POSITION(0,0)
                        ORIENT(-90)
        ENDREPEAT
ENDCOMPONENT

COMPONENT chip
WITH
        IClegs: iclegs1 WINDOW(135,15) AT(0,0,0)
                PINCOUNT(135,15,8,5)
        IClegs: iclegs2 WINDOW(135,15) AT(135,50,180)
                PINCOUNT(135,15,8,5)
ENDCOMPONENT
END
```

Figure 4: Model Definition for Circuit

4 Specific Tests

The inspection process is where the user sees the bulk of the work being done. All the previous work has been done to set up a framework ready for applying the actual inspections. The choice of tests to apply depends on what the user chooses. These are given in the model file and are associated with each feature that is to be inspected.

Each of the tests are independent of the others. No information is passed between them about the form of the feature being tested. This means that it is up to the user to give a sensible batch of tests to apply. Each of the tests should be designed to identify the presence of particular traits in the feature being inspected. These can vary widely from simple ones that might measure the length of some body to those that do a more complex analysis of the image such as finding the positions and counting the number of features.

The data that each test has to work with has a standardised form. For each feature there is an associated window within which it should be contained. The test requires only this area of the image plus any parameters that are necessary to describe the feature that it is trying to inspect. The parameters might describe the feature's length, orientation or relative illumination compared to its background. These parameters are test specific.

The success or failure of the tests shows whether a feature has been identified, found to be in its correct position and undamaged. This is why it is important to have a sensible choice of tests that will cover all eventualities that might result from incorrect assembly. Incorrect assembly can take the form of badly positioned or missing parts. Handling during assembly might also cause parts to be damaged. Failing a test cannot be used to identify the cause of the failure. A test may fail for any of the three reasons mentioned above: bad positioning, missing part and damaged part. The number of tests failed for a feature being inspected can be used to judge the cause but only to give a rough indication. The smaller the number of tests failed then the more likely that the part is only damaged rather than missing. A badly positioned part may pass tests that depend simply upon presence as compared to position within the window. The final result of identifying the cause of tests failing has to be left for the user to decide.

The window being tested is highlighted when using the display option. The window is also shown in the upper left hand corner of the screen. This addition allows the test routine to add any extra information to show how it is applying its test such as a scan line along which it is making some examination. Each of the tests are also left to identify the reason for the particular test failing. This removes the necessity for the main program to understand the workings of the tests. Both of these features, display access and error

Figure 5: View of IC Pins

feedback, are in keeping with the modularity of the program. It is easy to interchange the tests without affecting other parts of the program. To show how well the overall inspection process managed, a final display is shown with all the features that failed tests outlined. This allows the user to see at a glance how well everything has proceeded.

A test plan is outlined in section 5.2 for evaluating the effectiveness of the inspection process.

4.1 Pin Test

This particular test is designed to identify the number of pins that an IC has and also that none of them have been bent. This is really two separate tests but the intermediate data created doing one test is also used by the other test. It was therefore decided to combine them. To get a suitable method for achieving this it is necessary to look at an image corresponding to the pins of an IC. Figure 5 shows a view of eight pins on an integrated circuit.

If we take an intensity profile of a scanline travelling through all of the pins then it is clearly visible that the peaks in intensity are coincident with the pins (Figure 6). Figure 5 has a line down the centre of the image showing where the scanline is. The metal pins reflect much more light than the surroundings and so it would seem that counting the pins would involve finding peaks in the intensity. It is not quite that simple. The background behind the pins can be picked up as well as the pins. If there is any metal or other reflective material then that can cause unwanted peaks in intensity. To stop this fooling the count it is necessary to look for consecutive high values in the intensity profile. At least five consecutive units of a high intensity was used as a guide when testing the program. This is scaled into screen pixels. For five units with a scaling factor of two this would be ten pixels. The value can be varied at the discretion of the user by altering the model file.

As well as deciding how many consecutive pixels to look for it is also necessary to decide what value constitutes a peak in the intensity. It was decided to find the average intensity

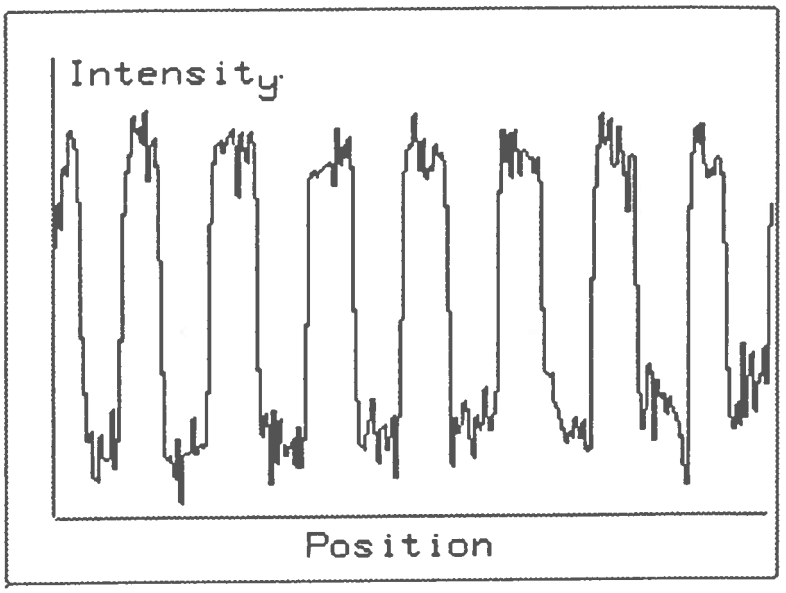

Figure 6: Intensity Profile Across Pins

along the scanline and add a small value on to that to give a threshold. The reason for using a threshold above the average is to try and stop noise from affecting the results. Since the pins are made of metal then they will reflect much more light and so will show as peaks well above the average rather than just above the average. A value of ten is added on to the average to give the threshold. This adds between seven and twelve percent on to the threshold depending upon the average intensity across the scan line. It is not necessary to allow this value to be changed since a different value would only be of advantage at either very low or very high average intensities. The high and low intensity profiles would be produced by either too much light or too little. In these conditions it would be too difficult to produce a reliable inspection process anyway.

Counting the pins now involves scanning across the window looking for five or more consecutive pixels above the threshold value. Each time such a peak[1] in intensity has been found a counter can be incremented.

Counting the pins is only half of the inspection process that is carried out. The other half involves checking to see that no pins are bent. This is a bit of a limiting case because there are many ways in which a pin could be bent. If a pin is bent back on itself then this

[1] A peak must have a beginning and an end. Therefore the counter is only incremented once the intensity drops below the threshold. This stops spurious peaks at the edges of the image window being picked up.

cannot be seen using any inspection method viewing from above. A pin bent straight out can be spotted. This is what is being looked for with this test. This is done by making two scans across the window. The first of these will be along where the pins should be. The second scan will be further out from the IC at the edge of the window. If there is a matching peak at the same position during both scans then it is assumed that a bent pin has been discovered.

To find matches it is necessary to store the positions in which pins are found. For each intensity peak found in the second scan there could be a possibility that it is a part of a bent pin. If there is a peak of the same size and in the same position in the first scan then it is assumed that a bent pin has been found. If the second scan was carried out part of the way along a track in the board then this would produce an intensity peak that should be ignored. Even though there may be a matching intensity peak at a pin position it is caused by an object that is much wider than a pin and so will be discarded.

A simple extension to the pin test would improve its reliablity. Using a straight line threshold copes with the ideal lighting situation where the intensity gradient across the image is negligible. If there is a large intensity gradient, which may be caused by a single light source close to the circuit board, then the present pin test will fail to work properly. To remedy this a least squares fit straight line could be used in place of the threshold value.

5 Test Plan

Once a working system has been developed it is necessary to comprehensively test the system. There now follows a test plan to achieve this aim.

There are two parts in the system which must be tested. The first of these is the model loader. It must be shown that the model data is correctly interpreted. The second part of the testing is to check that the specific inspection processes are carried out correctly.

5.1 Modelling Test Plan

Testing the model interpretation requires that the parsing of the model and subsequent modification of the parse tree be examined. This testing is aided by the inclusion of a run-time flag that will display the components within the circuit and how they are related. Looking at this output it is possible to see that the components have been properly linked together according to the model definition.

It is now necessary to define a series of models that will test the model loader. To do this each of the productions that define the grammar of the language for model descriptions must be tested. The grammar is shown in appendix A. The points that have to be considered in the grammar testing are discussed below in section 5.1.1. Example test models that cover the points discussed are given in section 5.1.2.

5.1.1 Testing The Model Loading

The top level production

$$grammar \ ::= \ list \ END \tag{1}$$

has only the one possible interpretation and so displays its validity every time that a model is parsed.

$$\begin{array}{lll} list & ::= & circuit \ | \\ & & component\_def \ | \\ & & circuit \ list \ | \\ & & component\_def \ list \end{array} \tag{2}$$

defines a list to contain a mixture of **circuit** and **component_def**, i.e. a mixture of circuit and component definitions. This can be tested by using combinations of more than one, one and zero of these instances. Zero instances should report an error since there will be insufficient information for creating a model.

$$\begin{aligned} circuit \;::=\; & CIRCUIT\ NAME\ WITH\ component\_list\ ENDCIRCUIT \;\mid \\ & CIRCUIT\ NAME\ WITH\ component\_list\ COMPARE \\ & inspection\_list\ ENDCIRCUIT \end{aligned} \tag{3}$$

Pairwise comparison tests have not yet been implemented but models can be created containing them. Therefore to test this production a dummy test has to be created that will inform the user that the test has executed. It will not actually carry out any inspections.

$$\begin{aligned} component\_list \;::=\; & repeatloop\ component\ ENDREPEAT \;\mid \\ & repeatloop\ component\ ENDREPEAT\ component\_list \;\mid \\ & component \;\mid \\ & component\ component\_list \end{aligned} \tag{4}$$

defines that a **component_list** can consists of one or more components and one or more components defined inside a repeat loop. Again this should be tested with zero, one or more instances of each type.

$$repeatloop \;::=\; REPEAT\ value\ STEP\ value\ ALONG\ NAME\ FOR \tag{5}$$

has three user defined parameters. The first **value** defines how many times the component should appear. The second **value** gives the spacing between components and the **NAME** shows which axis the components should be placed along. An error results if the axis name given does not begin with x or y.

The definition of **component** uses two names in its definition.

$$component \;::=\; COMPONENT\ NAME\ NAME\ posn \tag{6}$$

The first name relates to the component definition name. The second relates to the instance name. The component definition name is used to link the component instance with its definition since this is the only common factor between them. Correct usage can be shown by each component instance having the correct tests applied to it, i.e. the link between component instance and definition has been made correctly.

To show that the next two productions work as expected then it is necessary to see that the parameters read in are stored and used correctly.

$$\begin{array}{lll} posn & ::= & POSITION(value, value) \\ & & ORIENT(value) \\ window & ::= & WINDOW(value, value)\ AT(value, value, value) \end{array} \tag{7}$$

A model must be set up with appropriate parameter values so that the program will display a predictable action.

$$\begin{array}{lll} component\_def & ::= & COMPONENT\ NAME\ WITH\ feature\_list \\ & & ENDCOMPONENT \quad | \\ & & COMPONENT\ NAME\ WITH\ feature\_list \\ & & COMPARE\ inspection\_list ENDCOMPONENT \end{array} \tag{8}$$

This definition again requires the use of a dummy test for the same reasons as production 3 to show that the list of pairwise tests **inspection_list** is properly utilised.

The following two productions all have the same form:

$$\begin{array}{lll} feature\_list & ::= & feature \quad | \\ & & feature\ feature\_list \\ inspection\_list & ::= & inspection \quad | \\ & & inspection\ inspection\_list \end{array} \tag{9}$$

Testing these requires that zero, one or more instances of the items making up the list are included in the model file. Zero instances should not be accepted but any other number should be handled correctly.

A feature in a feature list is defined as the following:

$$\begin{aligned} feature \;::=\; & FEATURE: \; NAME \; window \; inspection\_list \;\; | \\ & repeatloop \; component \; ENDREPEAT \;\; | \\ & component \end{aligned}$$

(10)

There are three possibilities for a feature. It can be a predefined feature, a component within a repeat statement or a component.

Each predefined feature has a type *FEATURE* and a name associated with it. It is positioned according to the specification **window** and has a list of tests after it (**inspection_list**). It is necessary to check that the correct inspection tests are applied for the appropriate feature.

The component appearing as a feature can be tested by defining a subcomponent to appear inside another component. A component inside a repeat statement can be tested in the same way as a subcomponent except that there should be the designated number tested.

$$inspection \;::=\; TYPE \; (parm\_list)$$

(11)

defines either a feature test plus a number of parameters (10) or a pairwise test plus parameters (8) & (3). It is necessary to show that the correct parameters are passed to the particular test. The parameters can consist of a mixture of numbers and names. At present only the pairwise comparison tests would take names in the parameter list. Checking to see that the correct parameters are used with the test would require that the test print them out. This was done at the early development stage but has now been removed.

$$\begin{aligned} parm\_list \;::=\; & value \;\; | \\ & value \; ', ' \; parm\_list \;\; | \\ & NAME \;\; | \\ & NAME \; ', ' \; parm\_list \end{aligned}$$

(12)

Numbers can be either integers or floating point. They are stored as floating point so that the individual tests can use either form.

$$value ::= INTEGER \mid FLOAT \tag{13}$$

5.1.2 Example Test Models

There are two stages required for testing the model grammar. The first of these is to check that invalid model files will not be accepted. The second stage is to check that valid model files are correctly stored.

The type of invalid model files that will be discussed here cover the following features of the grammar:

1. Where a list of items is expected, none are given
2. Missing parameters given to inspection tests
3. Invalid feature types used
4. Missing component and circuit definitions

1. There are four cases where a list of items is expected. These involve the productions at 2, 4, 9 and 12. To show that these invalid cases are not accepted by the parser the following sections of model files could be included in a syntactically correct model file such as the one shown in figure 5.

   The first model is simply just the keyword **END**. This tests production rule 2.

   CIRCUIT memory WITH ENDCIRCUIT

   There is no list of components on the circuit board and so production 4 cannot accept this.

- COMPONENT memory WITH ENDCOMPONENT
- COMPONENT memory WITH feature_list COMPARE ENDCOMPONENT
- IClegs: p1 WINDOW(10,10) AT(0,0,0)

These are examples of a missing feature list for a component, missing test list for a pairwise comparison, and missing tests for a predefined feature (see production 9).

IClegs: p1 WINDOW(10,10) AT(0,0,0)
PINCOUNT()

This is an example of an empty parameter list which would not get past the parser. It is a slightly different case from missing parameters because just so long as there is at least one parameter then the parameter list will parse (production 12).

2. Missing parameters are checked for within production (11). Each type of test has a requirement for a certain number of parameters. This is checked against the actual number given before the parsing can continue.

3. There are four types of predefined feature: ICbody, IClegs, RESISTOR and CAPACITOR. Only these names can appear in the position designated for a predefined feature. The same follows for the test types.

4. Missing component and circuit definitions are not coped with at parse time. They can only be checked for once the model has been loaded in. The program will then look for the circuit definition that it is to used. If this is not found then an error is given. The same is done for each component that appears within the circuit definition. If one of these does not exist then an error is given. Since subcomponents can appear within components then for each component that appears in the circuit, all of its subcomponents must also exist. A model file such as the one shown in figure 5 will show what happens for missing definitions. Each component definition can in turn be removed from the file to show that its removal will trigger an error. The same can be done for the circuit definition.

Once the above testing has taken place it is necessary to look at how the inspection process uses a syntactically correct model definition. The following points are taken into consideration at this stage:

1. A correct circuit definition is specified if the circuit exists within model definition.
2. Components are correctly related to other components as subcomponents.

3. Each feature has the correct tests applied to it.
4. The positioning of the components appear as the model says they should.
5. Repeat statements work correctly.
6. Recursively defined components work correctly.

1. An optional check flag can be given when invoking the program (see appendix B). This will display which circuit is being inspected. It should be checked when there is one or more circuit definitions in the model.
2. The check flag will also invoke the program to display the component names in a roughly tree-like structure. This allows the user to see that all the components, subcomponents and features belong to the correct component definitions. The top level components (those that appear in the circuit definition) are shown leftmost in the output. If the model shown in figure 5 is used with the check flag on then the output would look like the following:

```
Circuit part list for memory:

memchip

        ic-a
                iclegs1
                iclegs2
        ic-b
                iclegs1
                iclegs2
        ic-c
                iclegs1
                iclegs2
        ic-d
                iclegs1
                iclegs2
res1

        r1
        r2
```

3. Each feature should have the correct tests applied to it. As the inspection process proceeds each feature name is displayed along with all the tests that are applied to it. This will show that production (10) is applied correctly along with the production

that called it (8), and the corresponding rule for the component instance (6) was linked correctly with the component's definition.

4. The positioning of the components (productions at (7)) requires a carefully designed model to show that the positions are as they should be. A model such as the one below is suitable.

```
CIRCUIT c1
WITH
        COMPONENT chip ic1
                POSITION(200,300)
                ORIENT(45)
ENDCIRCUIT
COMPONENT chip
WITH
        ICbody: b1 WINDOW(60,75) AT(0,10,-90)
                LENGTH(11,2)
ENDCOMPONENT
END
```

   Notice that all of the positional parameters are different from each other. This is to try and stop a coincidental error corrections should there be any incorrectly stored positional data. The area that is tested can be viewed on the framestore by setting the display flag (see appendix B). It can be compared with the expected results.

5. Repeat statements have three parameters (production (5)) that should be checked. The first of these is the number of times the component should be repeated. It is not allowed to be less than one as this would not make any sense. The number of times the repeat is carried out using an example file will show whether it is being used correctly. The step size between components is the next parameter. This has to be a visual test. The third parameter is the axis along which the components are placed. This can only be the x-axis or the y-axis. As well as checking that only these are accepted, a visual check should be made to see that the correct axis is being used.

6. The final language feature to test is the use of recursive subcomponents. It is necessary to see that the windowing system is using all the co-ordinate frames of each component correctly.

The model in figure 8 was used to carry out the testing discussed above. It has all the language features included that are necessary for testing. For all of the tests above the model was used with minor modifications as necessary such as trying different feature names, using different axes for the repeat statements etc.

```
CIRCUIT
WITH
        REPEAT 3 STEP 60 ALONG x FOR
                COMPONENT IC chip
                        POSITION(15,200)
                        ORIENT(-90)
        ENDREPEAT
        COMPONENT IC chip1
                POSITION(195,200)
                ORIENT(-90)
ENDCIRCUIT

COMPONENT IC
WITH
        IClegs: iclegs1 WINDOW(135,15) AT(0,0,0)
                PINCOUNT(135,15,8,5)
        IClegs: iclegs2 WINDOW(135,15) AT(135,50,180)
                PINCOUNT(135,15,8,5)
        REPEAT 2 STEP 10 ALONG y FOR
                COMPONENT resistor r1
                        POSITION(50,15)
                        ORIENT(0)
        ENDREPEAT
        COMPONENT resistor r2
                POSITION(-20,-30)
                ORIENT(90)
ENDCOMPONENT

COMPONENT resistor
WITH
        RESISTOR: r1 WINDOW(30,25) AT(0,0,0)
ENDCOMPONENT
END
```

Figure 8: Model Used For Testing Program

5.1.3 Model Testing Results

The tests outlined in the previous section were carried out using the model given. It was found that the program behaved as expected. It should be pointed out that the implementation does have limits built into it. The depth of subcomponent definitions is limited to nine. The maximum length of filenames is thirty characters. These limits will not be exceeded in normal circumstances.

5.2 Inspection Test Plan

The inspection test plan shows that the tests used during the inspection process are applied correctly under varying conditions. The varying conditions that will be examined are the translation and rotation of the circuit board within the image, and varying lighting conditions. Only variations in lighting should affect the reliability of the tests.

5.2.1 Translation and Rotation of Image

The program has been tested using the image shown in figure 4 as a reference. The next step is to show that the inspection process is invariant under translation and rotation of the image.

Applying the inspection process with the model file in figure 5 to the image translated should still get the same test results. The new circuit reference point is required in the command line (see appendix B). The same model should then be used with the image rotated about its centre. Finally both a translation and rotation should be tried.

5.2.2 Lighting Conditions

The previous test plan should not affect the results in any way. This next series of tests could cause the results to change. Using varying lighting conditions will show how durable the tests are. There are three lighting conditions to try which should give a feeling for how resilient the tests are.

Using daylight as the light source will give an even light cast over the circuit board without causing too much shadowing and no highlighting. A single spotlight on the circuit board will give a much brighter image than that achieved with daylight. It will cast shadows into the image since the light cannot be directly above all the components[2]. Using a second

[2]The image is taken by a camera directly above the centre of the circuit board and hence the light cannot

light from the opposite side of the circuit board will help reduce the shadowing and still give a bright image.

5.2.3 Inspection Test Plan Results

It was found that translating the circuit board within the image taken by the camera did not affect the results achieved during inspection except in one aspect. Translating the image, or rotating for that matter, may result in parts of the circuit board that are to be inspected not being visible. The program will reply with a "Window Outside Boundary" error in these cases. Those parts of the image which were visible were found to be unaffected by the translation.

Rotating the image did cause some problems. Due to rounding errors when working with floating point arithmetic and also due to the small error margins involved it was found that some tests failed. At this time only the one test is implemented. This is the pin test described in chapter 4. The scan line across the pins is only a few pixels wide. This results in the pincount test missing some pins when using the rotated image.

The same problem appeared when using the image that was both translated and rotated. No other problems were encountered due to anything other than inaccuracies caused by using a rotated image.

The aim of using different lighting conditions was not carried out to the full specifications due to restrictions in the time available. The following results would have been expected for the pin test if they had been carried out.

- **Daylight:** Since the variation in intensity across the circuit board would have been minimal then I would have expected the pin tests to work satisfactorily.
- **Single Light:** This was carried out at one angle where the pins were perpendicular to the direction of the light source. This minimalised the intensity variation across the scan lines through the pins. The results from the pin tests worked correctly. Since the pin test uses local information to decide upon a threshold (see chapter 4.1) then the average intensity variations between different sets of pins did not make any difference. If the scan lines across the pins had been in the direction of the light then there would have been a visible intensity gradient across the pins. The single threshold value used in the test would not have been able to cope with this. To cope with this a best line fit to the average intensity gradient would be required.

---

be positioned in the ideal place

- **Double Lighting System:** The problems with intensity gradients would be reduced when using two lights from opposite sides. The specular reflections would be intensified with the greater amount of light present. All the metal surfaces would reflect the light back and cause parts of the image to wash out. This effect can be seen slightly in the leftmost set of pins in figure 5.

6 Achievements and Conclusions

The aim of the project was to produce a reliable system for the inspection of the components on a variety of circuit boards. This required the development of a language which could be used to describe circuit boards. A framework was then built which could take the language descriptions of circuit boards and use them to guide an inspection process.

The grammar for the modelling language included a number of features that simplified the definition of models for describing circuits. The main features which extended the language beyond that originally developed by Chan [1] are:

- Referencing of components. Rather than repeatedly defining a component each time it is needed, a reference to it can be used.
- Hierarchical component definitions. Commonly occurring features can be grouped together into component definitions. These definitions can then be used inside other component definitions. This helps to keep the model readable.
- Repeat statements. If a number of the same component appear at regular intervals along the circuit board then they can be replaced by a single statement. This saves repetition during the modelling.

The development of a series of tests for feature recognition was not accomplished as had originally been anticipated. Too much time had been spent on the development of the program framework. This was caused by an unfamiliarity with the programming language used. One test was developed for the **ICpins** feature. The remaining features, **ICbody**, **CAPACITOR** and **RESISTOR**, do not yet have any tests designed for them.

The modularity of the program's framework lends itself well to an easily extendible system. The inspection mechanism which calls the individual tests does not know anything about these tests except the names of them. This means that the number of tests can be extended by increasing the list of tests which the inspection mechanism knows about. The grammar would also have to be altered slightly so that the test names would be recognised by the lexical analyser. It is also just as easy to extend the number of component features that can be used within the modelling language.

Two test plans were given which were used to evaluate the effectiveness of the program as a whole. The first test plan was designed to see that the model loading procedure was carried out correctly. This was found to be so. The second test plan was used to find out how resilient the feature tests were to varying conditions. These varying conditions consisted

of circuit boards which appeared at rotated and translated positions within the digitized image, and different lighting conditions. The program was found to work satisfactorily under rotations and translations. The different lighting conditions were not investigated. Only the one lighting condition was tried - a single light source to one side. This did highlight an improvement that could be made to the pin test (cf 4.1).

There is much work that can still be done on this project. It falls into three areas:

- Improving the modelling language.
- Developing more feature tests.
- Improving the interface.

Improvements have been suggested to the modelling language (cf 3.4). The first of these allows the user to abstract away from associating particular tests with features, and instead to use a more intuitive description. The second proposal is to introduce pairwise tests. The grammar used with the YACC parser already has the syntax for pairwise comparisons. The framework would have to be modified to call pairwise tests.

The main development that must be done concerns the feature tests. There are no tests which can be used for the **ICbody**, **CAPACITOR** and **RESISTOR** features. At present if these features appear in a model then the dummy test **LENGTH** must be added. The features will not be tested but if the display flag is set then the feature will be highlighted to show that they would have been tested had there been any to apply.

As it stands, the program applies tests and reports any failures when applying them to features. It does not attempt to reason why the failures occurred. It is up to the tests themselves to display helpful diagnostics for the user. It would be useful, for instance, if a misplaced part could be reported. This is something that is discussed by Barnard [4]. In his system he uses statistical tests to detect damaged, missing and misplaced parts.

There are two papers which have aspects that would be useful in this system. The first of these is a paper by Woods and Taylor [2]. They also use a model-driven system. One particular aspect is that the co-ordinate system is automatically calibrated. At present my program requires the user to give information relating to the position, orientation and scale of the circuit board in the image. Automatic calibration is also discussed in a paper by Bolles [3]. In it he describes the *local feature focus method* that is used when recognising partially visible objects. A reliable feature is used to determine a co-ordinate system from which other key features can be found so that an object's position and orientation can be determined. This would be useful if implemented to find the position and orientation of the

circuit board.

7 Acknowledgements

There are a number of people who have been of help to me during this project that I would like to thank:

- Bob Fisher, my supervisor, who helped with general aspects of how to approach the project and who also spent time reading over drafts of the project.
- Dave Croft who answered my queries about the graphical interface, and who managed to get the new framestore up and working just before the old one broke.
- Andrew Tait who read some early drafts of the report.

References

[1] Chan David *Grey Level Part Inspection*. Msc Thesis 1989

[2] P W Woods & C J Taylor *Application of a User-Programmable Vision System to Inspection of Complex Assemblies*. Proceeding of 5th Alvey Vision Conference p31-36 1989

[3] Bolles Robert C *Locating Partially Visible Objects: The Local Feature Focus Method*. AAAI National Conference on Artificial Intelligence p41-44 1980

[4] Barnard Stephen T *Automated Inspection Using Gray-Scale Statistics*. AAAI National Conference on Artificial Intelligence p49-52 1980

[5] Lex and Yacc *Sun© User Manual*

A Grammar BNF

The following section shows the BNF of the grammar for the modelling language described in chapter 3. Upper case words denote keywords which the lexical analyser recognises. There are three exceptions to this:

- **NAME** refers to any string of alpha-numeric characters.
- **INTEGER** and **FLOAT** refer to integer and floating point numbers respectively.

These three tokens are special cases for the lexical analyser used along with the parser. The parser is written using YACC and the following BNF grammar.

```
grammar          ::=  list END
list             ::=  circuit                                              |
                      component_def                                        |
                      circuit list                                         |
                      component_def list
circuit          ::=  CIRCUIT NAME WITH component_list ENDCIRCUIT          |
                      CIRCUIT NAME WITH component_list COMPARE
                      inspection_list ENDCIRCUIT
component_list   ::=  repeatloop component ENDREPEAT                      |
                      repeatloop component ENDREPEAT component_list        |
                      component                                            |
                      component component_list
repeatloop       ::=  REPEAT value STEP value ALONG NAME FOR
component        ::=  COMPONENT NAME NAME posn
posn             ::=  POSITION'('value','value')'
                      ORIENT'('value')'
component_def    ::=  COMPONENT NAME WITH feature_list
                      ENDCOMPONENT                                         |
                      COMPONENT NAME WITH feature_list
                      COMPARE inspection_list ENDCOMPONENT
feature_list     ::=  feature                                              |
                      feature feature_list
feature          ::=  FEATURE':' NAME window inspection_list               |
                      repeatloop component ENDREPEAT                       |
                      component
window           ::=  WINDOW'('value','value')' AT'('value','value",'value')'
inspection_list  ::=  inspection                                           |
                      inspection inspection_list
inspection       ::=  TYPE '('parm_list')'
parm_list        ::=  value                                                |
                      value ',' parm_list                                  |
                      NAME                                                 |
                      NAME ',' parm_list
value            ::=  INTEGER                                              |
                      FLOAT
```

B User Instructions

This program is designed to take a digitized image of a circuit board and apply an inspection process to it according to a model definition of what should be contained in the circuit board. It therefore requires two input files:

- Model Definition File
- Digitized Image File

Before the program can be run, the digitized image has to be in HIPS format and compressed from a 4:3 horizontal:vertical scaling to a 1:1 scaling. Some information also has to be given to the program about where the circuit board appears in the image. An (x,y) co-ordinate along with an orientation has to be stated. This will give the program a reference point relative to which the inspections will take place. This point has to be the same point which was used when the model file was created. It is therefore best to choose a prominent feature on the circuit board. As well as this it is necessary to give a scaling factor for converting from model co-ordinates to screen pixels.

There are two flags which can be invoked to get more feedback about what the program is doing. The first of these is the (c)heck flag. This will make the program print out a tree like diagram of how the components within the model relate to each other. The program does not run the tests when this flag is set. It will stop after the tree has been displayed. The second option is the (d)isplay flag. This will make use of a connected framestore to inform the user of how the inspection is proceeding. As each feature in the circuit is inspected, the area of the image that is being inspected will be inverted to show this. These areas of the screen are not necessarily going to be parallel with the x-axis of the display so another copy of the area is show in the top left hand corner of the screen. Any actions that a particular test may be doing can also be displayed in the top left hand portion of the screen.

The command to invoke the inspection program looks like the following:

ci -c -d modelfile circuit_name imagefile X Y Rot Scale

The -c and -d flags are optional arguments. The circuit_name is the circuit definition from the model file that is to be used for the inspection process. The arguments X Y and Rot define the reference point for the circuit board and take integer values. The scaling factor converting from model to screen co-ordinates is a floating point value.

C Program

The program consists of seventeen files. Here is a short description of what each one does:

- **Makefile** The bulk of this makefile is used for setting up the libraries required for using the framestore. Otherwise it is a standard makefile that will compile the following code fragments together.
- **ci.c** This is the main section of code which calls the various program sections as required. It reads in the command line, calls the parser to parse the model file and starts the control procedure.
- **ci.extern** All the external variables are defined in here.
- **ci.h** It contains some general constant definitions and all of the structures that are used by the YACC parser.
- **feed.c** is used by the windowing system. It rotates an area of the screen round until it is parallel with the axes of the screen.
- **halloc.c** contains a memory allocation function used by the windowing system when storing screen images.
- **length.c** contains a dummy inspection test which was used during the testing of the program.
- **lex.l** is the lexical analyser code which is written using LEX.
- **model.h** defines constants for each of the features and test names.
- **pin.c** is the pin test code.
- **pread.c** is a small piece of code for loading in images from a file. It stores them in a single dimensional array.
- **read_header.c** is used to load in the header information from a HIPS image in preparation for reading in the actual image.
- **show.c** This is used when the display flag is set. If any inspections of features fail then the showerrors function is called to display a box around the offending feature on the framestore.
- **test.c** The main inspection control mechanism resides here. The appropriate tests are called for each feature in the circuit.

- **wframe.c** is responsible for displaying the windows which signify the features being tested.
- **window.c** contains the mechanism which translates a feature's model co-ordinates to screen co-ordinates.
- **yacc.y** contains the grammar describing the modelling language. It is written using YACC.

C.1 Extending the Program

An important feature of this program is the ease with which it can be extended to include more inspection tests and features. There now follows a description of how to do this:

Adding new features to the language does not require much effort. This is because a feature is merely a label to which tests are associated. Therefore it is only necessary to give the program the name of the feature. This requires changes to two files - lex.l and model.h A new token has to be added to the lexical analyser in the form

Feature_name yylval.ival=Feature_name; return(FEATURE);

A constant definition for that feature has to be entered in the model.h file:

```
#define Feature_name <number>
```

where *number* is a numeric value that is not already used within the file. The nubmering system presently used is that features are numbered from 1000 up to 1999. Test names are numbered from 2000 upwards.

Adding new tests obviously requires a definition to be entered into the model.h file as mentioed above. A similar change is made to the lexical analyser, lex.l, except that TYPE is returned rather than FEATURE, i.e.

Test_name yylval.ival=Test_name; return(TYPE);

The parser, yacc.y, has to be informed of the new test. In the production

inspection ::= TYPE(parm_list)

there is a case statement that holds a list of all the tests. A new case has to be entered for the new test. The case statement is used to determine whether the correct number of arguments have been given to the test. So a new entry might look like the following for a new test **ORIENT**:

```
inspection:
        TYPE '(' parm_list ')'
        {
                arity=count($3);
                switch($1) {
                case PINCOUNT:
                        if (arity!=4)
                                error("Wrong Arity");
                        else
                                break;

                          :

                case ORIENT:
                        if (arity!=2)
                                error("Wrong Arity");
                        else
                                break;

                          :

                }
        if (($$=(INSP_PTR) malloc(sizeof(INSP)))==NULL)
                error("Can't allocate memory");
        $$->insptype=$1;
        $$->insparm=$3;
        }
;
```

Finally test.c has to be changed so that it will call the new test when it is found in the model definition. The function inspect() has a case statement that calls the appropriate test. A new case could be entered which calls the test function. The needs to take four arguments - the feature name being tested, the parameters that are required for the test to carry out the inspection, the scale for applying to any of the parameters that require it, and a display flag which informs the test if the framestore is available for use. The function also has to return a value (TRUE/FALSE or 1/0) which signifies whether the test succeeded. The new entry in the inspect function would look like the following:

```
switch(tst->insptype)
{
case PINCOUNT:
        ok=pincount(feature_name,tst->insparm,scale,display);
        break;
case ORIENT:
        ok=orient(feature_name,tst->insparm,scale,display);
break;

     :

default:
        ok=TRUE;
        break;
}
```

D Program Listing

```
/*************************************************************************/
/*              ci.c                                                     */
/*      This is the top level of the whole program                       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "hipl_format.h"
#include "ci.h"
#include "model.h"
#include "ci.extern"
#include "vision_box.h"
#include "string.h"

FILE *yyin;
FILE *image_fp;
char *sourcename;
char *imagefile;
char *circuit;
float scale;
struct header hd;
OBJ_LS_PTR object;

char *findcomponent();

main(argc,argv)
int argc;
char **argv;
{
        char filename[30];
        int count=0,row,col,display=FALSE,check=FALSE,x,y;
        COMP_LS_PTR component,feature;
        COMP_DEF_PTR defn;
        unsigned char *outdata;
        char *errorname;

        /* Read Parameters */

        if (argc==1)
                perr("usage: ci [-c][-d]<definitionsfile><circuitname><imagefile><X><Y><Rot><Scale>");

        if (argv[count+1][0]=='-')
                if (argv[count+1][1]=='c')
                {
                        count++;
                        check=TRUE;
                }
        if (argv[count+1][0]=='-')
                if (argv[count+1][1]=='d')
                {
                        count++;
                        display=TRUE;
                        vision_box_type=2;
                }

        if (argc<count+8)
                perr("usage: ci [-c][-d]<definitionsfile><circuitname><imagefile><X><Y><Rot><Scale>");

        sourcename=argv[count+1];
```



```
if ((yyin=fopen(sourcename,"r"))==NULL)
        error("can't open grammar file",sourcename);
if (yyparse()) exit(1);
fclose(yyin);

circuit=argv[count+2];
imagefile=argv[count+3];
Xref=atoi(argv[count+4]);
Yref=atoi(argv[count+5]);
rotation=atoi(argv[count+6]);
scale=atof(argv[count+7]);

if (rotation<-360 || rotation>360)
        perr("Invalid range for rotation");


/***** RESTRUCTURE PARSE TREE *****/

/**** Find appropriate circuit definition ****/

object=obj_head;

while (object!=NULL && object->circptr==NULL)
        object=object->objnext;

while (strcmp(object->circptr->circnm,circuit))
{
        object=object->objnext;
        while (object!=NULL && object->circptr==NULL)
                object=object->objnext;
        if (object==NULL)
                error("Circuit definition does not exist: ",circuit);
}


/**** Check that all necessary components are defined ****/

component=object->circptr->compls;

while (component!=NULL)
{
        errorname=findcomponent(&defn,component->compentry);
        if (!strcmp(errorname,""))
        {
              component->compentry->definition = defn;
              component=component->compnext;
        }
        else
        {

                printf("Component definition missing: %s\n\n",errorname);
                exit(-1);
        }
}


/**** Stores image file in array */

if ((image_fp=fopen(imagefile,"r"))==NULL)
```

```
        error("Cannot open image file",imagefile);
    fread_header(image_fp,&hd);
    if (hd.pixel_format!=PFBYTE)
    {
        fprintf(stderr,"Frame must be in byte format");
        exit(1);
    }
    for (row=0; row<hd.rows; row++)
    {
        if ((indata[row]=(unsigned char*)calloc(hd.cols,1))==NULL)
        {
            fprintf(stderr,"Cannot allocate array");
            exit(0);
        }
        if ((dest[row]=(unsigned char*)calloc(hd.cols,1))==NULL)
        {
            fprintf(stderr,"Cannot allocate array");
            exit(0);
        }
        pread(image_fp,indata[row],hd.cols);
        empty(dest[row],hd.cols);
    }
    if (display)    /* set up framestore for display. Show circuit image */
    {
        fsnum=1;
        vb_plane(1);
        vb_open();

        if ((outdata=(unsigned char *)malloc(hd.rows*hd.cols))==(unsigned char*)NULL)
        {
            fprintf(stderr,"Can't allocate array\n");
            exit(0);
        }
        for (y=0; y<hd.rows; y++)
            for (x=0; x<hd.cols; x++)
                outdata[x+y*hd.cols]=indata[y][x];
        vb_putblock(outdata,0,0,hd.cols,hd.rows);
        vb_plane(0);
        vb_putblock(outdata,0,0,hd.cols,hd.rows);
    }

    /**** Call to Control Routine ****/

    feature = object->circptr->compls;

    if (check) printf("Circuit part list for %s:\n\n",object->circptr->circnm);

    control(&feature,display,check);
    vb_plane(1);
    vb_close();
}


/*********************************************************************/
/* This function calls the test function for each component defined in */
/* the circuit being inspected.                                      */
/*********************************************************************/

control(feature,display,check)
```



```
COMP_LS_PTR *feature;
int display,check;
{
    COMP_LS_PTR feat;
    FEAT_LS_PTR subcomp;
    REPEAT_PTR rptinfo;
    int count,step,loop,repeat,len;
    char axis, *name;

    feat=(*feature);
    while (feat!=NULL)
    {
        if (feat->rpt==NULL)     /* simple component */
        {
            if (check)
                printf("%s\n\n",feat->compentry->compnm);
            subcomp=feat->compentry->definition->featls;
            while (subcomp!=NULL)
            {
                if (subcomp->feat->featnm==NULL)
                {
                    if (subcomp->feat->rpt == NULL)
                    {
                        repeat=FALSE,count=0;
                        step=0;axis=NULL;
                    }
                    else
                    {
                        rptinfo=subcomp->feat->rpt;
                        count=rptinfo->times;
                        step=rptinfo->step;
                        axis=rptinfo->axis[1];
                        repeat=TRUE;
                    }
                    subcontrol(&(subcomp->feat->compfeat),
                        repeat,axis,count,step,1,display,check);
                }
                else
                {
                    if (check)
                    {
                        printf("\t");
                        printf("%s\n",subcomp->feat->featnm);
                    }
                }
                subcomp=subcomp->featnext;
            }
            /* call to test component */
            if (!check)
                test(feat->compentry,hd.rows,hd.cols,scale,display);
        }
        else /* repeat component */
        {
            rptinfo=feat->rpt;
            count=rptinfo->times;
            step=rptinfo->step;
            axis=rptinfo->axis[1];
            len=strlen(feat->compentry->compnm);
            for (loop=1; loop<=count; loop++) {
```

```
                        feat->compentry->compnm[len-2]='-';
                        feat->compentry->compnm[len-1]='a'+loop-1;
                        feat->compentry->compnm[len]='\0';
                        subcontrol(&(feat->compentry),FALSE,NULL,0,0,0,display,check);
                        if (axis='x')
                                feat->compentry->wind[0]+=step;
                        else
                                feat->compentry->wind[1]+=step;
                }
        }
        feat=feat->compnext;
    }
}

/*********************************************************************/
/* This function will make a recursive call for any subcomponents    */
/* within the component. It also calls the test function for the     */
/* component.                                                        */
/*********************************************************************/

subcontrol(component,repeat,axis,count,step,tabs,display,check)
COMP_PTR *component;
int repeat,count,step,tabs,display,check;
char axis;
{
        FEAT_LS_PTR subcomp;
        REPEAT_PTR rptinfo;
        int loop,newrepeat,ncount,nstep,len;
        char naxis;

        if (!repeat)  /* simple component */
        {
                if (check)
                {
                        for (loop=0; loop<tabs; loop++)
                                printf("\t");
                        printf("%s\n",(*component)->compnm);
                }
                subcomp=(*component)->definition->featls;
                while (subcomp!=NULL) /* recurse down list of features */
                {
                        if (subcomp->feat->featnm==NULL) /* component found */
                        {
                                if (subcomp->feat->rpt == NULL) /* no repeat */
                                {
                                        naxis=NULL; ncount=0;
                                        nstep=0; repeat=FALSE;
                                }
                                else /* repeating component */
                                {
                                        rptinfo=subcomp->feat->rpt;
                                        naxis=rptinfo->axis[1];
                                        ncount=rptinfo->times;
                                        nstep=rptinfo->step;
                                        repeat=TRUE;
                                }
                                newrepeat=(subcomp->feat->rpt!=NULL);
                                subcontrol(&(subcomp->feat->compfeat),newrepeat,
                                        naxis,ncount,nstep,tabs+1,display,check);
```



```
                        }
                        else /* predefined feature found */
                        {
                                if (check)
                                {
                                        for (loop=0; loop<=tabs;loop++)
                                                printf("\t");
                                        printf("%s\n",subcomp->feat->featnm);
                                }
                        }
                        subcomp=subcomp->featnext;
                }
                /* call to test sub component */
                if (!check)
                        test(*component,hd.rows,hd.cols,scale,display);
        }
        else /* for repeat component have to feed in correct suffix for   */
             /* component and call function again for the number of times */
             /* given in repeat statement.                                */
        {
                len=strlen((*component)->compnm);
                for (loop=1; loop<=count; loop++) {
                        (*component)->compnm[len-2]='-';
                        (*component)->compnm[len-1]='a'+loop-1;
                        (*component)->compnm[len]='\0';
                        subcontrol(&(*component),FALSE,NULL,0,0,tabs,display,check);
                        if (axis='x')
                                (*component)->wind[0]+=step;
                        else
                                (*component)->wind[1]+=step;
                }
        }
}

/*********************************************************************/
/* This function looks through all the definitions in the model to find */
/* the one that it requires. If it succeeds then it returns the empty   */
/* string otherwise it will return the name of the component.           */
/*********************************************************************/

char *findcomponent(defn,component)
COMP_DEF_PTR *defn;
COMP_PTR component;
{
        OBJ_LS_PTR nextobject;
        COMP_DEF_PTR newcomp;
        FEAT_LS_PTR flist;
        char *name, *name2, *errorname;

        name=component->comptype;
        nextobject=obj_head;
        while (nextobject->componentptr == NULL) /* skip until component */
        {                                        /* definition found    */
                nextobject=nextobject->objnext;
                if (nextobject == NULL)
                        return(name);
        }
```

```
        /* search for component definition. If not found then return */
        /* from function call.                                       */

        while (strcmp(nextobject->componentptr->compdefnm,name))
        {
                nextobject=nextobject->objnext;
                if (nextobject == NULL)
                        return(name);
                while (nextobject->componentptr == NULL)
                {
                        nextobject=nextobject->objnext;
                        if (nextobject == NULL)
                                return(name);
                }
        }
        (*defn) = nextobject->componentptr;
        flist= (*defn)->featls;
        while (flist != NULL)
        {
                if (flist->feat->featnm == NULL)
                {
                        flist->feat->compfeat->parent=component;
                        errorname=findcomponent(&newcomp,flist->feat->compfeat);
                        if (!strcmp(errorname,""))
                        {
                                flist->feat->compfeat->definition=newcomp;
                                flist=flist->featnext;
                        }
                        else
                        {
                                name2=strcat(":",name);
                                name=strcat(errorname,name2);
                                return(name);
                        }
                }
                else flist=flist->featnext;
        }
        return("");
}

/*************************************************************************/
/* The following three functions are error routines. yyerror() is used */
/* by YACC, but the others are used throughout the code.               */
/*************************************************************************/

error(as,bs)
char *as;
char *bs;
{
        fprintf(stderr,"%s %s\n\n",as,bs);
        exit(-1);
}


yyerror(s)
char *s;
{
        fprintf(stderr,"%s\n",s);
}
```



```
perr(s)
char *s;
{
        fprintf(stderr,"%s\n",s);
        exit(-1);
}
```

```
/*************************************************************************/
/*      test.c                                                           */
/* test() takes a pointer to a component and calls all the appropriate */
/* tests for each feaure in the component.                              */
/*************************************************************************/


#include <stdio.h>
#include <string.h>
#include "ci.h"
#include "model.h"
#include "ci_extern"
#include "vision_box.h"

test(comp,rows,cols,scale,display)
COMP_PTR comp;
int rows,cols,display;
float scale;
{
        FEAT_LS_PTR features;
        INSP_LS_PTR inspection;
        int flag,x1,x2,y1,y2;
        int ok;
        char *name;
        COMP_PTR nameptr;
        struct coordinate vertex[4];

        features=comp->definition->featls;
        while (features != NULL)
        {
        /* get window for each feature in turn */

                flag=getwindow(rows,cols,features->feat->wind,scale,comp,vertex);
                if (flag==TRUE)
                {
                        printf("Window Outside Picture Boundary\n\n");
                        return;
                }
                /* rotate round window ready for applying tests */
                convert(rows,cols,features->feat->wind,scale,vertex);

                /* display highlighted window */
                if (display) wframe(rows,cols);

                inspection=features->feat->featinsp;

                /* call each test in turn for the feature in question */
                while (inspection != NULL)
                {
                        ok = inspect(features->feat->featnm,
                                inspection->testptr,rows,cols,scale,display);
                        if (!ok)
                        {
                                printf("\t *** Test failed for %s",features->feat->featnm);
                                nameptr=comp;
                                while (nameptr != NULL)
                                {
```



```
                                        printf(" %s",nameptr->compnm);
                                        nameptr=nameptr->parent;
                                }
                                printf("\n\n");
                                if (display) showerrors(vertex);
                        }
                        inspection=inspection->inspnext;
                }
                features=features->featnext;
        }
}


/*************************************************************************/
/* inspect() calls the appropriate test and gives it the parameters     */
/*************************************************************************/

int inspect(feature_name,tst,rows,cols,scale,display)
char *feature_name;
INSP_PTR tst;
int rows,cols,display;
float scale;
{
        int ok;
        PARM_LS_PTR parms;

        parms=tst->insparm;

        printf("testing on feature %s\n",feature_name);
        switch(tst->insptype)
        {
                case PINCOUNT:
                        ok=pincount(feature_name,tst->insparm,scale,display);
                        break;
                case LENGTH:
                        ok=length();
                        break;
                default:
                        ok=TRUE;
                        break;
        }
        return(ok);
}
```

```
/*************************************************************************/
/*      window.c                                                         */
/* This is the main part of the windowing system.  It converts feature   */
/* windows from model co-ordinates to screen co-ordinates.               */
/*************************************************************************/


#include <stdio.h>
#include <math.h>
#include "ci.h"
#include "ci.extern"


struct mat_data
{
        double x, y, orient;
};

void mult_mat();


/*************************************************************************/
/* getwindow takes the parameters for a feature window. It returns the */
/* screen coordinates of the window's corner points and also fills the */
/* global array 'scanline' with the boundary points of the window.     */
/*************************************************************************/

int getwindow(rows,cols,parms,scale,comp,vertex)
int rows, cols;
int *parms;
float scale;
COMP_PTR comp;
struct coordinate vertex[4];
{
        struct mat_data frame[10];
        int num,wind_x,wind_y,wind_orient;
        float transform[3][3];
        int i,j,y, x_temp[4];
        int *parm_ptr;
        float temp_x, temp_y, length, width;
        float sin_theta,cos_theta;
        struct linear_eqn {
                float m;
                float c;
                int   ok;
                struct coordinate ep1;
                struct coordinate ep2;
        } equation[4];


        /* get transformation matrix */
        parm_ptr=parms;
        num = getframes(comp,frame,scale);
        mult_mat(frame,num+1,transform);

        /* get window parameters */

        length = *(parm_ptr++) * scale;
```



```
width = *(parm_ptr++) * scale;
wind_x = *(parm_ptr++) * scale;
wind_y = *(parm_ptr++) * scale;
wind_orient = *(parm_ptr++);

sin_theta=sin(wind_orient * PI /180.0);
cos_theta=cos(wind_orient * PI / 180.0);

/* work out window corner points in screen coords. */

vertex[0].x=wind_x;
vertex[0].y=wind_y;
vertex[1].x=wind_x+cos_theta*length;
vertex[1].y=wind_y+sin_theta*length;
vertex[2].x=wind_x+cos_theta*length+sin_theta*width;
vertex[2].y=wind_y+sin_theta*length+cos_theta*width;
vertex[3].x=wind_x+sin_theta*width;
vertex[3].y=wind_y+cos_theta*width;

for (i=0; i<4; i++)
{
        temp_x =  vertex[i].x;
        temp_y =  vertex[i].y;

        vertex[i].x =
                  transform[0][0] * temp_x +
                  transform[0][1] * temp_y +
                  transform[0][2];
        vertex[i].y =
                  transform[1][0] * temp_x +
                  transform[1][1] * temp_y +
                  transform[1][2];
}


/*** find the 4 linear equations which model the ***/
/*** window boundary                               ***/
/*  This following piece of code was originally    */
/* written by David Chan.                          */

for(i=0; i<4; i++)
{
        j = (i+1) % 4;
        temp_x = vertex[j].x-vertex[i].x;
        temp_y = vertex[j].y-vertex[i].y;
        if (temp_x==0)
        {
                equation[i].ok = FALSE;
                equation[i].m = 0.0;
                equation[i].c = vertex[i].x;
        }
        else
        {
                equation[i].ok = TRUE;
                equation[i].m = temp_y / temp_x;
                equation[i].c = vertex[i].y-equation[i].m*vertex[i].x;
        }
        equation[i].ep1.x = vertex[i].x;
        equation[i].ep1.y = vertex[i].y;
        equation[i].ep2.x = vertex[j].x;
```

```
        equation[i].ep2.y = vertex[j].y;
}

/*** Find the y-min and y-max coordinate of the window corners ***/

y_min=y_max=equation[0].ep1.y;
for (i=0; i<4; i++)
{
        if (equation[i].ep1.y > y_max)
                y_max = equation[i].ep1.y;
        if (equation[i].ep2.y > y_max)
                y_max = equation[i].ep2.y;
        if (equation[i].ep1.y < y_min)
                y_min = equation[i].ep1.y;
        if (equation[i].ep2.y < y_min)
                y_min = equation[i].ep2.y;
}

/*** Remove those linear equations which do not intersect ***/
/*** the current line                                     ***/

x_min=x_max=equation[0].ep1.x;
for (i=0; i<4; i++)
{
        if (equation[i].ep1.x > x_max)
                x_max = equation[i].ep1.x;
        if (equation[i].ep2.x > x_max)
                x_max = equation[i].ep2.x;
        if (equation[i].ep1.x < x_min)
                x_min = equation[i].ep1.x;
        if (equation[i].ep2.x < x_min)
                x_min = equation[i].ep2.x;
}
if(x_min < 0 || x_max > cols || y_min < 0 || y_max > rows)
        return(1);


/*** find the boundary points of the window by finding the ***/
/*** intersection points of the curret scan line with the  ***/
/*** 4 linear equations                                    ***/

for(y=y_min; y<y_max+1; y++)
{
        for(i=0,j=0; i<4; i++)
        {

                if((y>=equation[i].ep1.y&&y<=equation[i].ep2.y)||
                        (y<=equation[i].ep1.y&&y>=equation[i].ep2.y))
                {
                        if(equation[i].ok == FALSE)
                                x_temp[j++] = equation[i].c;
                        else {
                                if (equation[i].m == 0.0)
                                {
                                        x_temp[j++] = equation[i].ep1.x;
                                        x_temp[j++] = equation[i].ep2.x;
                                        break;
                                }
                                else
```



```
                                        x_temp[j++]=(y-equation[i].c)/equation[i].m;
                                }
                        }
                }
                scanline[y].left=scanline[y].right=x_temp[0];
                for(i=0; i<j; i++)
                {
                        if(x_temp[i] < scanline[y].left)
                                scanline[y].left = x_temp[i];
                        else if(x_temp[i] > scanline[y].right)
                                scanline[y].right = x_temp[i];

                }
        }
        return(0);
}


/*************************************************************************/
/* The co-ordinate frames required for the component are gathered        */
/* together here.                                                        */
/*************************************************************************/

int getframes(comp,frameinfo,scale)
COMP_PTR comp;
struct mat_data *frameinfo;
float scale;
{
        int number=9,loop;
        COMP_PTR component;

        /* get components coordinate frame */
        component=comp;
        frameinfo[number].x = (double) component->wind[0] * scale;
        frameinfo[number].y = (double) component->wind[1] * scale;
        frameinfo[number].orient = (double) component->wind[2];
        number--;

        /* get each supercomponents coordinate fame */
        while (component->parent != NULL)
        {
                frameinfo[number].x = (double)
                        component->parent->wind[0] * scale;
                frameinfo[number].y = (double)
                        component->parent->wind[1] * scale;
                frameinfo[number].orient = (double)
                        component->parent->wind[2];
                component=component->parent;
                number--;
                if (number<1)
                        error("Recursive Definition too deep for ",comp->compnm);
        }
        frameinfo[number].x= Xref;
        frameinfo[number].y= Yref;
        frameinfo[number].orient= rotation;

        /* have to copy coordinate frame information so that it */
        /* starts at index 0 in the array                       */
```

```
        for (loop=number; loop<10; loop++)
                frameinfo[loop-number]=frameinfo[loop];

        return(9-number); /* number of frames found */
}


/*************************************************************************/
/* This function takes all of the coordinate frames, forms matrices     */
/* out of them and multiplies them all together to obtain a single      */
/* transformation matrix.                                               */
/*************************************************************************/

void mult_mat(frames, numb, comp_mat)
struct mat_data *frames;
int numb;
float comp_mat[3][3];
{
        int h, i, j;
        float c, s, x, y, tmp_mat[3][3];

        for (i = 0; i < 3; i++)
        {
                for (j = 0; j < 3; j++)
                {
                        comp_mat[i][j] = 0.0;
                }
                comp_mat[i][i] = 1.0;
        }
        for (h = 0; h < numb; h++)
        {
                c = cos((frames[h].orient)*(PI/180.0));
                s = sin((frames[h].orient)*(PI/180.0));
                x = frames[h].x;
                y = frames[h].y;
                tmp_mat[0][0] = c*comp_mat[0][0]+s*comp_mat[0][1];
                tmp_mat[0][1] = c*comp_mat[0][1]-s*comp_mat[0][0];
                tmp_mat[0][2] = x*comp_mat[0][0]+y*comp_mat[0][1]+
                                comp_mat[0][2];
                tmp_mat[1][0] = c*comp_mat[1][0]+s*comp_mat[1][1];
                tmp_mat[1][1] = c*comp_mat[1][1]-s*comp_mat[1][0];
                tmp_mat[1][2] = x*comp_mat[1][0]+y*comp_mat[1][1]+
                                comp_mat[1][2];
                tmp_mat[2][0] = c*comp_mat[2][0]+s*comp_mat[2][1];
                tmp_mat[2][1] = c*comp_mat[2][1]-s*comp_mat[2][0];
                tmp_mat[2][2] = x*comp_mat[2][0]+y*comp_mat[2][1]+
                                comp_mat[2][2];
                for (i = 0; i < 3; i++)
                        for (j = 0; j < 3; j++)
                                comp_mat[i][j] = tmp_mat[i][j];
        }
}
```



```
/*************************************************************************/
/*      feed.c                                                           */
/* This module will take the window defined in the scanline array and    */
/* put it into a block array in a suitable form for applying tests       */
/*************************************************************************/

#include "stdio.h"
#include "math.h"
#include "ci.h"
#include "model.h"
#include "ci_extern"


/*************************************************************************/
/* this function produces a pointer to the array containing the area     */
/* of the screen to be used for inspection purposes.                     */
/*************************************************************************/

convert(rows,cols,parm_ptr,SF,vertex)

int *parm_ptr;
float SF;
struct coordinate vertex[4];
{
        int wind_x, wind_y, wind_orient, length, width;
        int destx,desty,refpt_x,refpt_y;
        int x1,y1,x2,y2;
        double X,Y,dist,rot;

        /* Get reference line for calculating angle of window */

        x1 = vertex[0].x; x2 = vertex[1].x;
        y1 = vertex[0].y; y2 = vertex[1].y;

        /* get window parameters */

        length = *(parm_ptr++);
        width = *(parm_ptr++);
        wind_x = *(parm_ptr++);
        wind_y = *(parm_ptr++);
        wind_orient = *(parm_ptr++);

        X = (x2-x1);
        Y = (y2-y1);
        if (X==0)
                rot=PI/2;
        else
                rot = atan2(Y, X);

        test_length=length*SF;
        test_width=width*SF;

        map(rows,cols,test_length,test_width,x1,y1,rot);
}

/*************************************************************************/
/* This function actually maps the window defined by x,y,width,length    */
/* and orient to the origin with the window length parallel to the       */
```

```
/* x-axis of the framestore display.                                    */
/************************************************************************/

map(rows,cols,length,width,x,y,orient)

int length,width,x,y;
double orient;
{
        int a,b,xloop,yloop,xpos,ypos;
        double cos_theta,sin_theta;

        cos_theta=cos(orient);
        sin_theta=sin(orient);

        for (yloop=0;yloop<width;yloop++)
        {
                for (xloop=0,xloop<length;xloop++)
                {
                        a=x+ cos_theta*xloop - sin_theta*yloop;
                        b=y+ sin_theta*xloop + cos_theta*yloop;

                        if (b<512 && b>=0 && a<512 && a>=0)
                                dest[yloop][xloop]=indata[b][a];
                }
        }
}
```



```
/*******************************************************/
/* File: wframe.c                                      */
/* This is the program which is responsible for        */
/* displaying the highlighted feature windows that are */
/* being tested.                                       */
/*******************************************************/

#include <stdio.h>
#include <hipl_format.h>
#include "vision_box.h"
#include <math.h>
#include "ci.h"
#include "ci_extern"

extern struct header hd;

wframe(rows, cols)

int rows, cols;

{
        unsigned char *outdata;
int i, strength=1;
        int left, right;
        int x, y;
        long picsize;

picsize=cols*rows;
        strength = 255;

        if ((outdata=(unsigned char *)malloc(picsize))==(unsigned char*)NULL)
        {
                fprintf(stderr,"can't allocate array\n");
                exit(0);
        }

        /*** initialize the frame ***/

        for (y=0;y<rows;y++)
        {
                for (x=0;x<cols;x++)
                        outdata[x+y*cols]=indata[y][x];
        }

        /*** set all the pixels between the two intersection points ***/
        /*** Left and Right of the current scanline line to 255     ***/

        for(y=y_min; y<=y_max; y++)
        {
               left = scanline[y].left;
               right = scanline[y].right;
               for(x=left; x<right; x++)
                        outdata[x+y*cols]=255-indata[y][x];
        }

        /* invert area of screen where window is to be displayed */

        for (y=0; y<test_width; y++)
```

```
    {
            for (x=0; x<test_length; x++)
                    outdata[x+y*cols]=dest[y][x];
    }
    for (y=0; y<test_width; y++)
            outdata[test_length+y*cols]=255;
    for (x=0; x<test_length; x++)
            outdata[x+test_width*cols]=255;

    vb_putblock(outdata,0,0,cols,rows);
    free(outdata);
}
```



```
/*************************************************************************/
/*      show.c                                                           */
/* showerrors() is used to display a rectangular window outline on the */
/* framestore's second image plane. This is used to build up an image  */
/* containing all the components that failed a test outlined           */
/*************************************************************************/


#include <stdio.h>
#include "vision_box.h"
#include "ci.h"
#include "model.h"
#include "ci_extern"


showerrors(vertex)
struct coordinate vertex[4];
{
        int x0,y0,x1,y1,x2,y2,x3,y3;

        fsnum=1;
        vb_plane(fsnum);

        x0=vertex[0].x; x1=vertex[1].x; x2=vertex[2].x; x3=vertex[3].x;
        y0=vertex[0].y; y1=vertex[1].y; y2=vertex[2].y; y3=vertex[3].y;

        if ((x0<x1) || ((x0==x1) && (y0<=y1))) vb_line(x0,y0,x1,y1,255);
        else vb_line(x1,y1,x0,y0,255);

        if ((x1<x2) || ((x1==x2) && (y1<=y2))) vb_line(x1,y1,x2,y2,255);
        else vb_line(x2,y2,x1,y1,255);

        if ((x2<x3) || ((x2==x3) && (y2<=y3))) vb_line(x2,y2,x3,y3,255);
        else vb_line(x3,y3,x2,y2,255);

        if ((x3<x0) || ((x3==x0) && (y3<=y0))) vb_line(x3,y3,x0,y0,255);
        else vb_line(x0,y0,x3,y3,255);

        fsnum=0;
        vb_plane(fsnum);
}
```

```
/*************************************************************************/
/*      pin c                                                            */
/* This file contains the pin test code.                                 */
/*************************************************************************/

#include "ci.h"
#include "ci.extern"
#include "vision_box.h"

#define THRESH=10
/*************************************************************************/
/* pincount() will count the number of intensity peaks along a scan      */
/* line which goes through the middle of the feature window. It then     */
/* compares the peak positions with those found on a second scan line    */
/* going across the bottom of the window. If any matches are found       */
/* then it is assumed that a bent pin has been found.                    */
/*************************************************************************/

pincount(name,parms,scale,display)
char *name;
PARM_LS_PTR parms;
float scale;
int display;
{
        int scan_ypos,pincount=0,average=0,threshold;
        int pin_pos[512],nopin_pos[512];
        int length,width,count,size,loop,temp1,temp2;
        int ok=TRUE;

        /* get paramters required by test and scale those that require it */

        length = (parms->parm)*scale;
        parms=parms->parmnext;
        width = (parms->parm)*scale;
        parms=parms->parmnext;
        count = parms->parm;
        parms=parms->parmnext;
        size = parms->parm*scale;

        /* work out average intensity along first scan line */

        scan_ypos=width/2;

        for (loop=0; loop<length; loop++)
        {
                if (display) vb_putpix(loop,scan_ypos,255);
                average+=dest[scan_ypos][loop];
        }
        average=average/length;
        threshold=average+THRESH;

        /* make first scan */

        pincount = pinscan(length,scan_ypos,threshold,size,pin_pos);

        if (pincount != count)
        {
                ok=FALSE;
                printf("Pin count failed: count=%d\n\n",pincount);
```



```
        }
        else
                printf("Pin count succeeded\n\n");

        /* make second scan at bottom of window */

        pincount = pinscan(length,width-1,threshold,size,nopin_pos);

        /* compare peak positions from two scans */
        /* If size of peak and position match then give warning */

        loop=0;
        while (loop<length)
        {
                if (pin_pos[loop]==TRUE)
                {
                        if ( nopin_pos[loop]==TRUE)
                        {
                                temp1=temp2=loop;
                                do {
                                        temp1++;
                                } while (pin_pos[temp1]==TRUE);
                                do {
                                        temp2++;
                                } while (nopin_pos[temp2]==TRUE);
                                if (abs(temp1-temp2)<2)
                                {
                                        if (display) vb_line(loop,0,loop,width-1,255);
                                        printf("Possible Bent Pin at pos %d\n\n",loop);
                                        ok=FALSE;
                                }
                                loop = (temp1>temp2) ? temp1 : temp2;
                        }
                        else if (nopin_pos[loop+1] != TRUE)
                        {
                                do {
                                        loop++;
                                } while (pin_pos[loop]==TRUE);
                        }
                        else
                                loop++;
}
                else
                 loop++;

        }
        return(ok);
}


/*************************************************************************/
/* This function makes the scan along the position given to it in ypos   */
/* If any peaks are found then they are stored in the array peakpos      */
/* The peak has to last for at least 'size' pixels and be above the      */
/* intensity value of threshold.                                         */
/*************************************************************************/

int pinscan(xsize,ypos,threshold,size,peakpos)
int xsize,ypos,threshold,size,peakpos[512];
```

```
{
        int loop,peak=0,pincount=0;

        for (loop=0; loop<xsize; loop++)
        {
                peakpos[loop]=FALSE;
                if (dest[ypos][loop]>threshold)
                {
                        peak++;
                        if (peak>=size) peakpos[loop]=TRUE;
                }
                else
                {
                        if (peak>=size) pincount++;
                        peak=0;
                }
        }
        return(pincount);
}
```



```
/*************************************************************************/
/*      length.c                                                         */
/*                                                                       */
/*      This is a dummy inspection test which was used during the        */
/* testing of the program. It prints out a message and then terminates   */
/*************************************************************************/

#include "ci.h"

int length()
{
        printf("Calling length test: Not yet implemented\n");
        return(TRUE);
}
```

```
/*************************************************************************/
/*      halloc.c                                                         */
/*************************************************************************/


/* Copyright (c) 1982 Michael Landy, Yoav Cohen, and George Sperling

Disclaimer:  No guarantees of performance accompany this software,
nor is any responsibility assumed on the part of the authors.  All the
software has been tested extensively and every effort has been made to
insure its reliability.   */

/*
 * halloc.c - HIPL Picture Format core allocation
 *
 * Michael Landy 2/1/82
 */

#include <stdio.h>

char *halloc(i,j)

int i,j;

{
char *k;

k = (char *) calloc(i,j);
if(k == NULL)
perr("Not enough core available.");
return(k);
}
```



```
/*************************************************************************/
/*      pread.c                                                          */
/* This contains functions for reading a byte format file(image) into    */
/* an array.  This was taken from David Chan's code.                     */
/*************************************************************************/

#include <stdio.h>

pread(fd,buf,count)
FILE *fd;
char *buf;
int count;
{
        int cnt,r;

        cnt=count;
        while (cnt>0)
        {
                *(buf++)=getc(fd);
                cnt--;
        }
        return(count-cnt);
}


empty(buf,count)
char *buf;
int count;
{
        int cnt

        cnt=count;
        while (cnt>0)
        {
               *(buf++)='\0';
                cnt--;
        }
        return(count-cnt);
}
```

```
/************************************************************************/
/*      read_header.c                                                   */
/* This is used to read the header information on an image file.  This  */
/* was not written by me.                                               */
/************************************************************************/

#include "hipl_format.h"
#include <stdio.h>
#define LINES 100
#define LINELENGTH 132

static char *ssave[LINES];
static int slmax[LINES];
static int lalloc = 0;
extern char *halloc();

fread_header(fd,hd)

struct header *hd;
FILE *fd;

{
        int lineno, len, i;
        char *s;
        char *getline();

        if(lalloc<1) {
                ssave[0] = halloc(LINELENGTH, sizeof (char));
                slmax[0] = LINELENGTH;
                lalloc = 1;
        }
        getline(fd,&ssave[0],&slmax[0]);
        hd->orig_name = halloc(strlen(ssave[0])+1, sizeof (char));
        strcpy(hd->orig_name,ssave[0]);
        getline(fd,&ssave[0],&slmax[0]);
        hd->seq_name = halloc(strlen(ssave[0])+1, sizeof (char));
        strcpy(hd->seq_name,ssave[0]);
        hd->num_frame = dfscanf(fd);
        getline(fd,&ssave[0],&slmax[0]);
        hd->orig_date = halloc(strlen(ssave[0])+1, sizeof (char));
        strcpy(hd->orig_date,ssave[0]);
        hd->rows = dfscanf(fd);
        hd->cols = dfscanf(fd);
        hd->bits_per_pixel = dfscanf(fd);
        hd->bit_packing = dfscanf(fd);
        hd->pixel_format = dfscanf(fd);
        lineno = 0;
        len = 1;
        getline(fd,&ssave[0],&slmax[0]);
        s = ssave[0];
        while(*(s += strlen(s)-3) == '|') {
                len += strlen(ssave[lineno]);
                lineno++;
                if (lineno >= LINES)
                        perr("Too many lines in header history");
                if(lineno >= lalloc) {
                        ssave[lineno] = halloc(LINELENGTH, sizeof (char));
                        slmax[lineno] = LINELENGTH;
```



```
                        lalloc++;
                }
                getline(fd,&ssave[lineno],&slmax[lineno]);
                s = ssave[lineno];
        }
        len += strlen(ssave[lineno]);
        hd->seq_history = halloc(len, sizeof (char));
        hd->seq_history[0] = '\0';
        for (i=0;i<=lineno;i++)
                strcat(hd->seq_history,ssave[i]);
        lineno = 0;
        len = 1;
        while(strcmp(getline(fd,&ssave[lineno],&slmax[lineno]),"\n")) {
                len += strlen(ssave[lineno]);
                lineno++;
                if (lineno >= LINES)
                        perr("Too many lines in header desc");
                if(lineno >= lalloc) {
                        ssave[lineno] = halloc(LINELENGTH, sizeof (char));
                        slmax[lineno] = LINELENGTH;
                        lalloc++;
                }
        }
        hd->seq_desc = halloc(len, sizeof (char));
        *hd->seq_desc = '\0';
        for (i=0;i<lineno;i++)
                strcat(hd->seq_desc,ssave[i]);
}

char *getline(fd,s,l)

char **s;
FILE *fd;
int *l;

{
        int i,m;
        char c,*s1,*s2;

        i = 0;
        s1 = *s;
        m = *l;
        while((c = getc(fd)) != EOF && c != '\n') {
                if (m-- <= 2) {
                        s2 = halloc(LINELENGTH+*l,sizeof (char));
                        strcpy(s2,*s);
                        *s = s2;
                        *l += LINELENGTH;
                        m = LINELENGTH;
                        s1 = s2 + strlen(s2);
                }
                *s1++ = c;
        }
        if (c == '\n') {
                *s1++ = '\n';
                *s1 = '\0';
                return(*s);
        }
        perr("Unexpected EOF while reading header.");
```

```
}

dfscanf(fd)

FILE *fd;

{
        int i;

        getline(fd,&ssave[0],&slmax[0]);
        sscanf(ssave[0],"%d",&i);
        return(i);
}
```



```
/*************************************************************************/
/*      ci.h                                                             */
/*      All the structures that are used by YACC for creating the        */
/* parse tree are defined here. There are a few general definitions too  */
/*************************************************************************/

#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define PI 3.141592654
#define POSN_ENTRIES 3
#define WIN_ENTRIES 5
#define ROWS 512

typedef struct parm_ls {
        float parm;
        char *name;
        struct parm_ls *parmnext;
        } PARM_LS;

typedef PARM_LS *PARM_LS_PTR;


typedef struct insp {
        int insptype;
        PARM_LS_PTR insparm;
        } INSP;

typedef INSP *INSP_PTR;

typedef struct insp_ls {
        INSP_PTR testptr;

        struct insp_ls *inspnext;
        } INSP_LS;

typedef INSP_LS *INSP_LS_PTR;

typedef struct feat {
        int feattype;
        char *featnm;
        int wind[WIN_ENTRIES];
        INSP_LS_PTR featinsp;
        struct comp *compfeat;
        struct repeat *rpt;
        } FEAT;

typedef FEAT *FEAT_PTR;

typedef struct feat_ls {
        FEAT_PTR feat;

        struct feat_ls *featnext;
        } FEAT_LS;

typedef FEAT_LS *FEAT_LS_PTR;
```

```
typedef struct pair {
        char *name;
        int value;
        struct pair *pairnext;
        } PAIR;

typedef PAIR *PAIR_PTR;


typedef struct comp_defn {
        char *compdefnm;
        FEAT_LS_PTR featls;
        INSP_LS_PTR testls;
        } COMP_DEF;

typedef COMP_DEF *COMP_DEF_PTR;


typedef struct repeat {
        char *axis;
        int times,step;
        } REPT;

typedef REPT *REPEAT_PTR;

typedef struct comp {
        char *comptype;
        char compnm[50];
        PAIR_PTR pairls;
        int wind[POSN_ENTRIES];
        COMP_DEF_PTR definition;
        struct comp *parent;
        } COMP;

typedef COMP *COMP_PTR;

typedef struct comp_ls {
        COMP_PTR compentry;
        REPEAT_PTR rpt;
        struct comp_ls *compnext;
        } COMP_LS;

typedef COMP_LS *COMP_LS_PTR;

typedef struct circ {
        char *circnm;
        COMP_LS_PTR compls;
        INSP_LS_PTR testls;
        } CIRC;

typedef CIRC *CIRC_PTR;


typedef struct obj_ls {
        CIRC_PTR circptr;
        COMP_DEF_PTR componentptr;

        struct obj_ls *objnext;
        } OBJ_LS;
```



```
typedef OBJ_LS *OBJ_LS_PTR;


OBJ_LS_PTR obj_head;

struct rasterscan {
        int left;
        int right;
        } scanline[ROWS];

struct bound {
        int x;
        int y;
        };

unsigned char *indata[ROWS];
unsigned char *dest[ROWS];

int Xref,Yref;
int rotation;
int x_min,x_max,y_min,y_max;
int test_length,test_width;
```

```
/*************************************************************************/
/*      model.h                                                          */
/* Any new features or test names have to be put here.                   */
/* Feature names are numbered from 1000-1999.                            */
/* Test names are numbered from 2000 upwards                             */
/*************************************************************************/

#define ICbody          1000
#define IClegs          1001
#define CAPACITOR       1002
#define RESISTOR        1003
#define LENGTH          2000
#define SEPARATION      2001
#define RELORIENT       2002
#define PINCOUNT        2003
```



```
/*************************************************************************/
/*      yacc.y                                                           */
/* This is the YACC grammar file.                                        */
/*************************************************************************/

%{
#include "ci.h"
#include "ctype.h"
#include "model.h"
#include <stdio.h>
#include <string.h>
int i,arity;
OBJ_LS_PTR obj_head;
char *malloc();
char name[50];
%}

%start grammar
%union {
        int ival;
        float fval;
        char *symval;
        CIRC_PTR circ;
        OBJ_LS_PTR objt_ls;
        COMP_PTR comp;
        COMP_LS_PTR comp_ls;
        COMP_DEF_PTR comp_defn;
        REPEAT_PTR repeat;
        FEAT_LS_PTR feat_ls;
        FEAT_PTR feat;
        INSP_LS_PTR insp_ls;
        INSP_PTR insp;
        PARM_LS_PTR parm_ls;
        int place[POSN_ENTRIES];
        int win[WIN_ENTRIES];
        }

%token <fval> FLOAT
%token <ival> INTEGER FEATURE TYPE
%token <symval> NAME
%token END CIRCUIT ENDCIRCUIT COMPONENT ENDCOMPONENT POSITION ORIENT
%token COMPARE WITH WINDOW AT REPEAT STEP FOR ALONG ENDREPEAT

%type <objt_ls> list
%type <circ> circuit
%type <comp_ls> component_list
%type <comp> component
%type <comp_defn> component_def
%type <repeat> repeatloop
%type <feat_ls> feature_list
%type <feat> feature
%type <insp_ls> inspection_list
%type <insp> inspection
%type <parm_ls> parm_list
%type <fval> value
%type <win> window
%type <place> posn
%%
```

```
grammar:
        list END
        {
        obj_head=$1;
        }
;

list:
        circuit
        {
        if (($$=(OBJ_LS_PTR) malloc(sizeof(OBJ_LS)))==NULL)
                error("Can't allocate memory");
        $$->circptr=$1;
        $$->componentptr=NULL;
        $$->objnext=NULL;
        }
        |
        component_def
        {
        if (($$=(OBJ_LS_PTR) malloc(sizeof(OBJ_LS)))==NULL)
                error("Can't allocate memory");
        $$->componentptr=$1;
        $$->circptr=NULL;
        $$->objnext=NULL;
        }
        |
        circuit list
        {
        if (($$=(OBJ_LS_PTR) malloc(sizeof(OBJ_LS)))==NULL)
                error("Can't allocate memory");
        $$->circptr=$1;
        $$->componentptr=NULL;
        $$->objnext=$2;
        }
        |
        component_def list
        {
        if (($$=(OBJ_LS_PTR) malloc(sizeof(OBJ_LS)))==NULL)
                error("Can't allocate memory");
        $$->componentptr=$1;
        $$->circptr=NULL;
        $$->objnext=$2;
        }
;

circuit:
        CIRCUIT NAME WITH component_list ENDCIRCUIT
        {
        if (($$=(CIRC_PTR) malloc(sizeof(CIRC)))==NULL)
                error("Can't allocate memory");
        $$->circnm=$2;
        $$->compls=$4;
        $$->testls=NULL;
        }
        |
        CIRCUIT NAME WITH component_list COMPARE inspection_list ENDCIRCUIT
        {
        if (($$=(CIRC_PTR) malloc(sizeof(CIRC)))==NULL)
```



```
                error("Can't allocate memory");
        $$->circnm=$2;
        $$->compls=$4;
        $$->testls=$6;
        }
;

component_list:
        repeatloop component ENDREPEAT
        {
        if (($$=(COMP_LS_PTR) malloc(sizeof(COMP_LS)))==NULL)
                error("Can't allocate memory");
        $$->rpt=$1;
        $$->compentry=$2;
        $$->compnext=NULL;
        }
        |
        repeatloop component ENDREPEAT component_list
        {
        if (($$=(COMP_LS_PTR) malloc(sizeof(COMP_LS)))==NULL)
                error("Can't allocate memory");
        $$->rpt=$1;
        $$->compentry=$2;
        $$->compnext=$4;
        }
        |
        component
        {
        if (($$=(COMP_LS_PTR) malloc(sizeof(COMP_LS)))==NULL)
                error("Can't allocate memory");
        $$->compentry=$1;
        $$->compnext=NULL;
        $$->rpt=NULL;
        }
        |
        component component_list
        {
        if (($$=(COMP_LS_PTR) malloc(sizeof(COMP_LS)))==NULL)
                error("Can't allocate memory");
        $$->compentry=$1;
        $$->compnext=$2;
        $$->rpt=NULL;
        }
;

repeatloop:
        REPEAT value STEP value ALONG NAME FOR
        {
        if (($$=(REPEAT_PTR) malloc(sizeof(REPT)))==NULL)
                error("Can't allocate memory");
        $$->times=$2;
        if ($2<1) error("Cannot repeat component less than once");
        $$->axis=$6;
        if ((strcmp($6,"x")!=0) && (strcmp($6,"y")!=0))
        {
        printf("%d,%d \n",strcmp($6,"x"),strcmp($6,"y"));
                error("Invalid Axis for repeat:",$6);
        }
        $$->step=$4;
```

```
        }
;

component:
        COMPONENT NAME NAME posn
        {
        if (($$=(COMP_PTR) malloc(sizeof(COMP)))==NULL)
                error("Can't allocate memory");
        $$->comptype=$2;
        for(i=0; i<POSN_ENTRIES; i++)
                $$->wind[i]=$4[i];
        strcpy(name,$3);                /* Have to add an extra two      */
        strcat(name,"  ");               /* characters so that repeat     */
        strcpy($$->compnm,name);         /* components can have extended  */
        $$->pairls=NULL;                 /* names                         */
        $$->parent=NULL;
        }
;

posn:
        POSITION'('value','value')'
        ORIENT'('value')'
        {
        $$[0]=$3;
        $$[1]=$6;
        $$[2]=$9;
        }
;


component_def:
        COMPONENT NAME WITH feature_list ENDCOMPONENT
        {
        if (($$=(COMP_DEF_PTR) malloc(sizeof(COMP_DEF)))==NULL)
                error("Can't allocate memory");
        $$->compdefnm=$2;
        $$->featls=$4;
        $$->testls=NULL;
        }
        |
        COMPONENT NAME WITH feature_list COMPARE inspection_list ENDCOMPONENT
        {
        if (($$=(COMP_DEF_PTR) malloc(sizeof(COMP_DEF)))==NULL)
                error("Can't allocate memory");
        $$->compdefnm=$2;
        $$->featls=$4;
        $$->testls=$6;
        }
;

feature_list:
        feature
        {
        if (($$=(FEAT_LS_PTR) malloc(sizeof(FEAT_LS)))==NULL)
                error("Can't allocate memory");
        $$->feat=$1;
        $$->featnext=NULL;
        }
        |
```



```
        feature feature_list
        {
        if (($$=(FEAT_LS_PTR) malloc(sizeof(FEAT_LS)))==NULL)
                error("Can't allocate memory");
        $$->feat=$1;
        $$->featnext=$2;
        }
;

feature:
        FEATURE':' NAME window inspection_list
        {
        if (($$=(FEAT_PTR) malloc(sizeof(FEAT)))==NULL)
                error("Can't allocate memory");
        $$->featnm=$3;
        $$->compfeat=NULL;
        $$->featinsp=$5;
        for(i=0; i<WIN_ENTRIES; i++)
                $$->wind[i]=$4[i];
        }
        |
        repeatloop component ENDREPEAT
        {
        if (($$=(FEAT_PTR) malloc(sizeof(FEAT)))==NULL)
                error("Can't allocate memory");
        $$->featnm=NULL;
        $$->rpt=$1;
        $$->compfeat=$2;
        }
        |
        component
        {
        if (($$=(FEAT_PTR) malloc(sizeof(FEAT)))==NULL)
                error("Can't allocate memory");
        $$->featnm=NULL;
        $$->compfeat=$1;
        $$->rpt=NULL;
        }
;

window:
        WINDOW'('value','value')' AT'('value','value','value')'
        {
        $$[0]=$3;
        $$[1]=$5;
        $$[2]=$9;
        $$[3]=$11;
        $$[4]=$13;
        }
;

inspection_list:
        inspection
        {
        if (($$=(INSP_LS_PTR) malloc(sizeof(INSP_LS)))==NULL)
                error("Can't allocate memory");
        $$->testptr=$1;
        $$->inspnext=NULL;
        }
```

```
        |
        inspection inspection_list
        {
        if (($$=(INSP_LS_PTR) malloc(sizeof(INSP_LS)))==NULL)
                error("Can't allocate memory");
        $$->testptr=$1;
        $$->inspnext=$2;
        }
        |
        error
        {yyerrok;}
;

inspection:
        TYPE '(' parm_list ')'
        {
        arity=count($3);
        switch($1) {
        case SEPARATION:
        case RELORIENT:
        case PINCOUNT:
                if (arity!=4)
                        error("Wrong Arity");
                else
                        break;
        case LENGTH:
                if (arity!=2)
                        error("Wrong Arity");
                else
                        break;
        default:
                error("Invalid test");
        }
        if (($$=(INSP_PTR) malloc(sizeof(INSP)))==NULL)
                error("Can't allocate memory");
        $$->insptype=$1;
        $$->insparm=$3;
        }
;

parm_list:
        value
        {
        if (($$=(PARM_LS_PTR) malloc(sizeof(PARM_LS)))==NULL)
                error("Can't allocate memory");
        $$->parm=$1;
        $$->name=NULL;
        $$->parmnext=NULL;
        }
        |
        value ',' parm_list
        {
        if (($$=(PARM_LS_PTR) malloc(sizeof(PARM_LS)))==NULL)
                error("Can't allocate memory");
        $$->parm=$1;
        $$->name=NULL;
        $$->parmnext=$3;
        }
        |
```



```
        NAME
        {
        if (($$=(PARM_LS_PTR) malloc(sizeof(PARM_LS)))==NULL)
                error("Can't allocate memory");
        $$->parm=0;
        $$->name=$1;
        $$->parmnext=NULL;
        }
        |
        NAME ',' parm_list
        {
        if (($$=(PARM_LS_PTR) malloc(sizeof(PARM_LS)))==NULL)
                error("Can't allocate memory");
        $$->parm=0;
        $$->name=$1;
        $$->parmnext=$3;
        }
;

value:
        INTEGER = {$$=$1;}
        |
        FLOAT = {$$=$1;}
;

%%

#include "lex.c"


count(head)
PARM_LS_PTR head;
{
        if (head==NULL)
                return(0);
        else
                return(1+count(head->parmnext));
}
```

```
/*****************************************************************/
/*      lex.l                                                    */
/* This is the lexical analyser written using lex                */
/*****************************************************************/


%{
double atof();

int yytmpchr;
# undef input
# define ngetc() (yytmpchr=getc(yyin),putc(yytmpchr,stdout),yytmpchr)
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr):ngetc())==10?(yylineno++,yytchar):yytchar)==EOF?

#define NAMELENGTH 10000
char names[NAMELENGTH];
char *nameend = names;

char *makename();
%}
%%
CIRCUIT                 return(CIRCUIT);
ENDCIRCUIT              return(ENDCIRCUIT);
COMPONENT               return(COMPONENT);
ENDCOMPONENT            return(ENDCOMPONENT);
REPEAT                  return(REPEAT);
STEP                    return(STEP);
FOR                     return(FOR);
ALONG                   return(ALONG);
ENDREPEAT               return(ENDREPEAT);
COMPARE                 return(COMPARE);
POSITION                return(POSITION);
ORIENT                  return(ORIENT);
WITH                    return(WITH);
WINDOW                  return(WINDOW);
AT                      return(AT);
END                     return(END);
ICbody                  {yylval.ival = ICbody; return(FEATURE);}
IClegs                  {yylval.ival = IClegs; return(FEATURE);}
CAPACITOR               {yylval.ival = CAPACITOR; return(FEATURE);}
RESISTOR                {yylval.ival = RESISTOR; return(FEATURE);}
LENGTH                  {yylval.ival = LENGTH; return(TYPE);}
PINCOUNT                {yylval.ival = PINCOUNT; return(TYPE);}
SEPARATION              {yylval.ival = SEPARATION; return(TYPE);}
RELORIENT               {yylval.ival = RELORIENT; return(TYPE);}
[A-Za-z][A-Za-z0-9_]*   {yylval.symval = makename(yytext); return(NAME);}
-?".".*[0-9]+           {yylval.fval=atof(yytext); return(FLOAT); }
-?[0-9]+".".*[0-9]*     {yylval.fval=atof(yytext); return(FLOAT); }
-?[0-9]+                {yylval.ival=atoi(yytext); return(INTEGER); }
"/*" {/* delete comments */
 int c;
 do {
while (input() != '*') {};
c = input();
if (c == '/') break;
unput(c);
} while (1);
```



```
}
[ \t\n] ;
. return((int) *yytext);
%%

char *makename(p)
char *p;
{
        char *start;

        start = nameend;
        while (*p != '\0')
                {*nameend++ = *p++;
                 if (nameend >= names + NAMELENGTH)
                        error("name table overflow");
                }
        *nameend++ = '\0';
        return(start);
}
```