1991

A compact representation of phase diagrams

Ali Yildirim
Lehigh University

A Compact Representation of Phase Diagrams

by

Ali Yıldırım

A Thesis

Presented to the Graduate Committee

of Lehigh University

In Candidacy for the degree of

Master of Science

in

Computer Science

Lehigh University

1991

This thesis is accepted and approved in fulfillment of the requirements for the degree of Master of Science in Computer Science.

May 2, 1991
Date

D. K. Decker
Advisor in Charge

L. J. Vanecek
CSEE Department Chairperson

Acknowledgments

I would like to thank Prof. Donald Hillman and Prof. Richard Decker for their direction when I was lacking it and for their being patient with my work. I also would like to thank my country for financially supporting my education in the USA.

Table of Contents

List of Figures

1. Abstract

In today's industrial world, alloys, which play a very important role in electronics industry in producing and packaging integrated circuits, are broadly used. Since phase diagrams represent changes of state involving one or more substances and are broadly used in packaging integrated circuits , in this work, we are looking for a way of compactly representing phase diagrams in the computer, displaying them on the screen, and retrieving information from them to help engineers working in integrated circuits packaging.

As a result, we developed a compact representation that requires less than 2K bytes of memory to store a typical phase diagram. Using this representation, we are able to display phase diagrams on the screen and to make inferences on them using a rule-based expert system.

Introduction

Many industrial operations, such as the manufacture of ceramics, alloys, glass, and salts, depend on knowledge of phase diagrams which are graphical representations of phase changes involving one, two, or more substances while they are being cooled or heated. Many industries, including the steel industry, and naturally, the electronics and computer industry etc., make use of alloys. Since phase diagrams represent information about phase changes of alloys, many industries depend on this knowledge. The use of alloys in semiconductor electronics for production of integrated circuits is essential. The alloys are mainly used in semiconductor electronics for package assembly and to attach a die to the package. Adhesives could also be used for this purpose but they are not as good at conducting the heat. One of the most important issues in manufacturing integrated circuits is the problem of heat dissipation from the integrated circuit. To avoid problems due to heat dissipation, it is better to use alloys instead of adhesives to attach the die to the package. Alloys are also used to attach the lid as well as the pins to the package. Another use of alloys is to attach a flip chip to a multicircuit package. When performing this, the flip chip solder bumps shrink. When attaching different parts of an integrated circuit to the package, alloys with different melting temperatures are used since we do not want previously

used alloys to remelt while another alloy is being used at its melting temperature to attach a part to the package. In other words, we do not want to undo something which was done earlier. Therefore, for instance, if a braze, which is a high-temperature solder, is used to attach the die to the package, a low-temperature solder is used to attach the lid to the package.

As mentioned earlier, phase changes involving one or more substances are represented by phase diagrams. A human expert in phase diagrams can easily interpret the information contained in the diagram. However, this process of visual interpretation cannot be as easily done by computers as by humans. In order for the computer to interpret phase diagrams like people do, the phase diagrams must be represented in the computer by a standard representation. A paper written by Don J. Orser [1] says that it is possible to represent phase diagrams using topological structures, inherent in the phase diagrams, and explains a way of doing so. According to him, in order to answer queries regarding phase adjacencies, the topological structure must be explicitly represented in the computer.

I definitely agree with him. Therefore, what we are looking for is a compact representation of phase diagrams, other than topological structures, in order to be able to display them on the screen and to provide answers to queries made by a knowledge based system.

2. Overview of Phase Diagrams

In terms of a material's microstructure, a phase is a region which differs in structure and composition from another region. Solid and liquid are just two of the possible phases. A phase diagram is a graphical representation of phase changes of one or more substances undergoing changes in pressure and temperature or in some other combination of variables such as solubility and temperature. Lead-tin, silver-copper, and indium-antimony are three of the most known phase diagrams.

Since phase changes are so much a part of the physical world, phase diagrams are pertinent to everyday existance: the melting of ice, the boiling of water, the formation of fog, the setting of cement, for instance, are well-known phase changes. In the laboratory, processes such as extraction, crystallization, distillation, precipitation, the use of freezing mixtures, and the identification of substances all involve phase changes describable by phase diagrams.

Phase diagrams contain experimental data. With any system, the first step of experimental study is to establish what phases result from different compositions of the given substances under a variety of specific conditions. The results of the experiments are then plotted and embodied in phase diagrams.

Phase diagrams can also be calculated from theoretical data as well. Computer programs have been developed to compute phase boundaries between two regular solution phases, and

phase boundaries between compound solution phases. Most of the calculated phase diagrams agree materially with observations.

Phase diagrams are used by engineers and scientists to understand and predict many aspects of the behavior of materials. Some of the important information obtainable from phase diagrams is as follows:

1. the phases present at different compositions and temperatures under slow cooling conditions
2. the equilibrium solid solubility of one element or compound in another
3. the temperature at which an alloy cooled under equilibrium conditions starts to solidify and the temperature range over which solidification occurs
4. the temperature at which different phases start to melt

Phase diagrams are generally classified by the number of components involved in the system, such as unary, binary, and ternary systems.

2.1. Unary Systems

The simplest phase diagrams are the ones for unary systems. When only one component is present, every possible phase is 100 percent of that component. The two variables of pressure and temperature suffice to determine the state of the system.

A pure substance such as water can exist in solid, liquid, and vapor phases, depending on the conditions of temperature and pressure. A very familiar example of two phases of a pure substance is a glass of water containing some ice. In this case, solid and liquid water are two distinct phases which are separated by a phase boundary, the surface of the ice cubes. During the boiling of water, liquid water and water vapor are two phases in equilibrium. Below is the graphical representation of the phases of water.

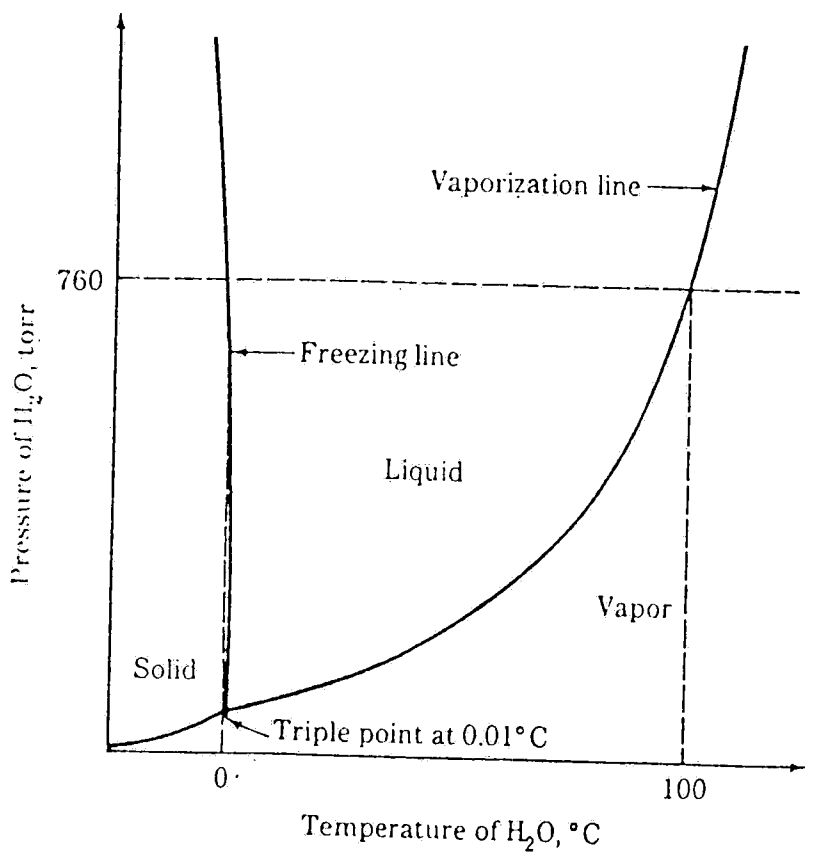

FIGURE 1 Approximate phase diagram for pure water

At any point in the regions delineated by the curves, the pressure and temperature allow only one phase, solid, liquid, or vapor, to exist. At any point on the curves, two phases are allowed to exist: solid-liquid, solid-vapor, and liquid-vapor. Along the line between solid and liquid, the melting

temperatures for different pressures can be found. In this phase diagram, there exists a triple point at a low pressure and low temperature where solid, liquid, and vapor phases can coexist. The triple point represents the unique conditions under which all three phases can coexist in equilibrium.

2.2. Binary Systems

A mixture of two metals is called a binary alloy and constitutes a two-component system. In this case, when the pressure is kept constant, the maximum number of phases which can coexist in equilibrium is three. This three-phase equilibrium takes place only at an invariant point. There are five kinds of binary systems: isomorphous, eutectic, peritectic, monotectic, and complex systems.

2.2.1. Isomorphous Systems

In some binary metallic systems, the two elements are completely soluble in each other in both the liquid and solid states. In these systems, only a single type of crystal structure exists for all compositions of the components; therefore, they are called isomorphous systems.

An important example of isomorphous binary alloy systems is the copper-nickel system (fig.2). The area above the upper line in the diagram, called the liquidus, corresponds to the region of stability for the liquid phase. On the other hand, the area below the lower line, called the solidus, corresponds

to the region of stability for the solid phase. A two-phase region where the liquid and solid phases can coexist is represented by the area between the solidus and liquidus in which the amount of each phase present depends on the temperature and chemical composition of the alloy.

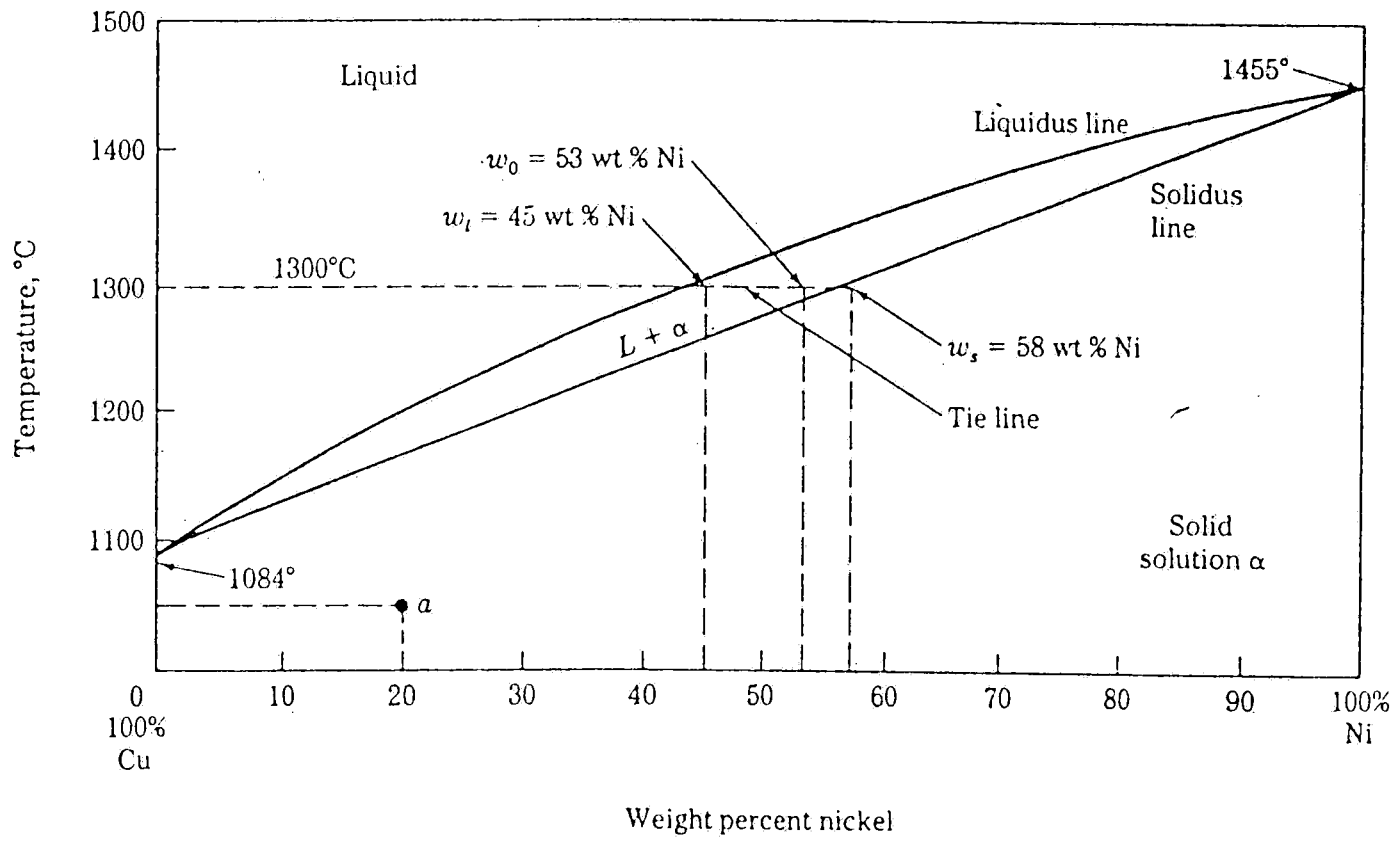

FIGURE 2 The copper-nickel phase diagram

2.2.2. Eutectic Systems

Many binary alloy systems have components which have limited solid solubility in each other. This type of binary system usually forms two solid solutions with limited solubilities. A typical binary eutectic phase diagram is the one for lead-tin alloy, shown in fig.3. In this diagram, there are three one-phase regions (α), (β), and (liquid), three two-phase regions (α+β), (α+liquid), and (β+liquid), and one three-phase region. The regions of restricted solid

solubility, α and β phases, are called terminal solid solutions. Unlike the β phase, which is a tin-rich solid solution, the α phase is a lead-rich solid solution.

In simple binary eutectic systems, there is a specific alloy composition known as the eutectic composition which freezes at a lower temperature than all other compositions.

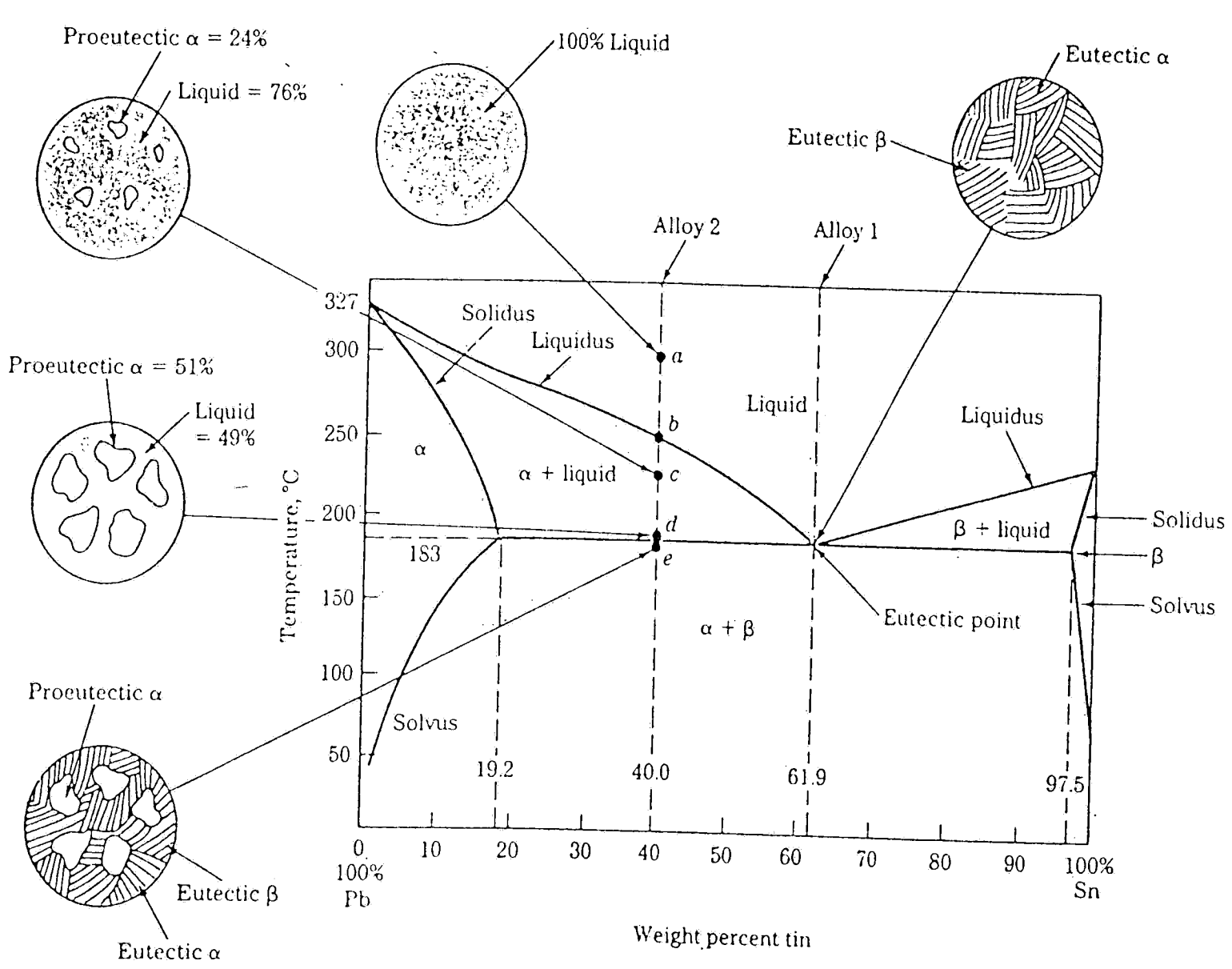

FIGURE 3 The lead-tin equilibrium phase diagram

This low temperature which corresponds to the lowest temperature at which the liquid phase can exist is called the eutectic temperature which is 183 °C for the lead-tin system. The eutectic composition is 61.9 weight percent tin and 38.1

weight percent lead.

Eutectic systems have an invariant reaction, called the eutectic reaction, in which, when cooled slowly, the liquid phase transforms into two different solid phases.

2.2.3. Peritectic Systems

Another type of reaction that frequently occurs in binary phase systems is the peritectic reaction in which a liquid phase reacts with a solid phase to form a new and different solid phase. This reaction is commonly present as part of more complicated binary systems, particularly if the melting points of the two components, under constant pressure, are quite different.

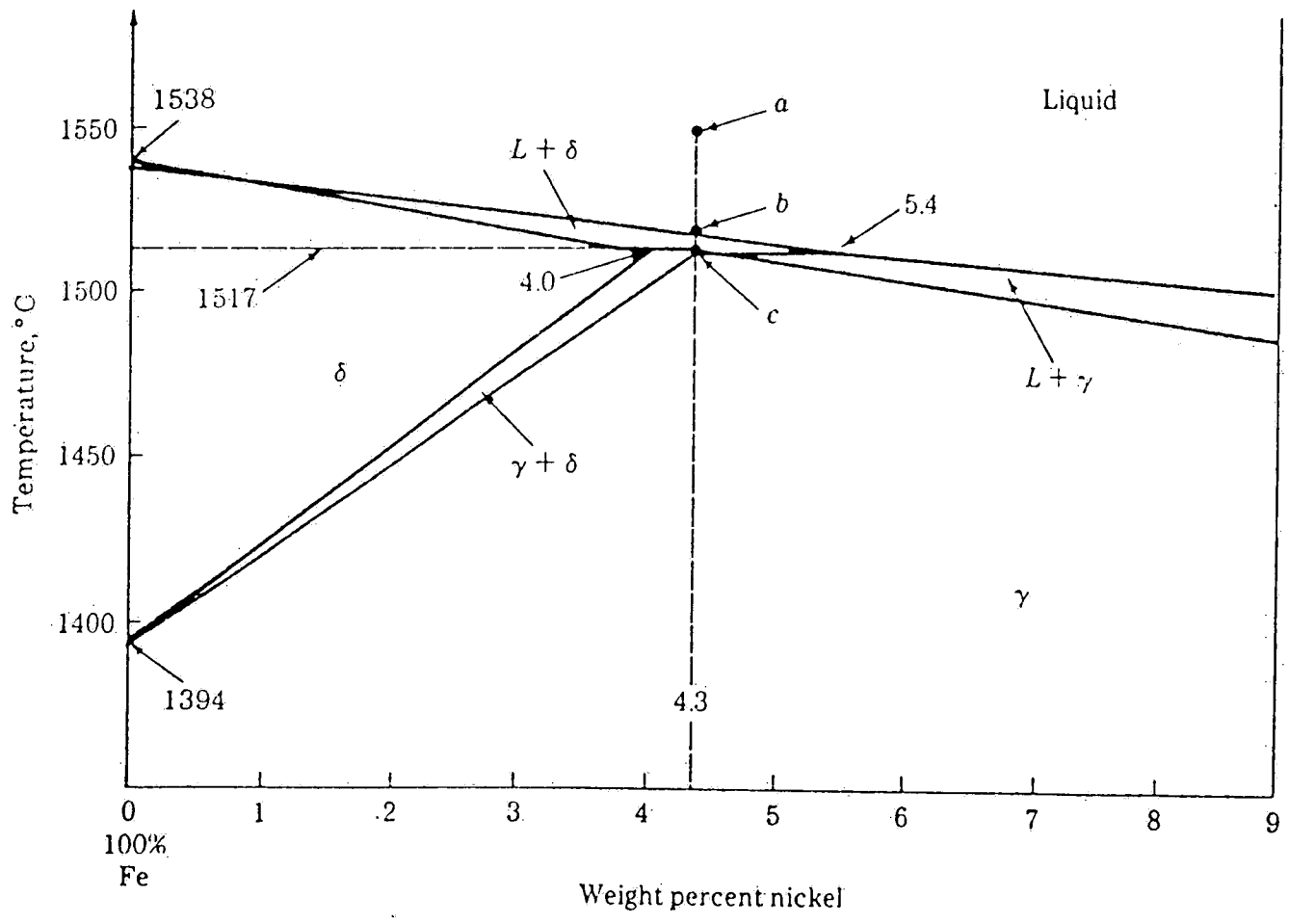

FIGURE 4 The peritectic region of the iron-nickel phase diagram

The peritectic reaction of the iron-nickel phase diagram is shown in fig.4. In this diagram, there are two solid phases, δ and γ, and one liquid phase. The δ phase is a solid solution of nickel in BCC iron, whereas the γ phase is a solid solution of nickel in FCC iron. The BCC iron and FCC iron are two forms of iron with different structures. The peritectic point c is defined by the peritectic temperature of 1517 °C and the peritectic composition of 4.3 weight percent nickel in iron. This point is invariant because three phases δ, γ, and liquid can coexist in equilibrium.

2.2.4. Monotectic Systems

The third type of three-phase reaction is the monotectic reaction in which a liquid phase transforms into a solid phase and another liquid phase. These two liquids are immiscible, just like oil and water, and therefore constitute individual phases. A reaction of this type occurs in the copper-lead system, whose phase diagram is shown in fig.5., at 955 °C and 36 weight percent lead in copper. Either a eutectic or a peritectic reaction can appear in the lower temperature region; the former is more often encountered. The copper-lead phase diagram has an eutectic point at 326 °C and 99.94 weight percent lead, and as a result terminal solid solutions of almost pure lead (0.007% copper) and pure copper (0.005% lead) are formed at room temperature.

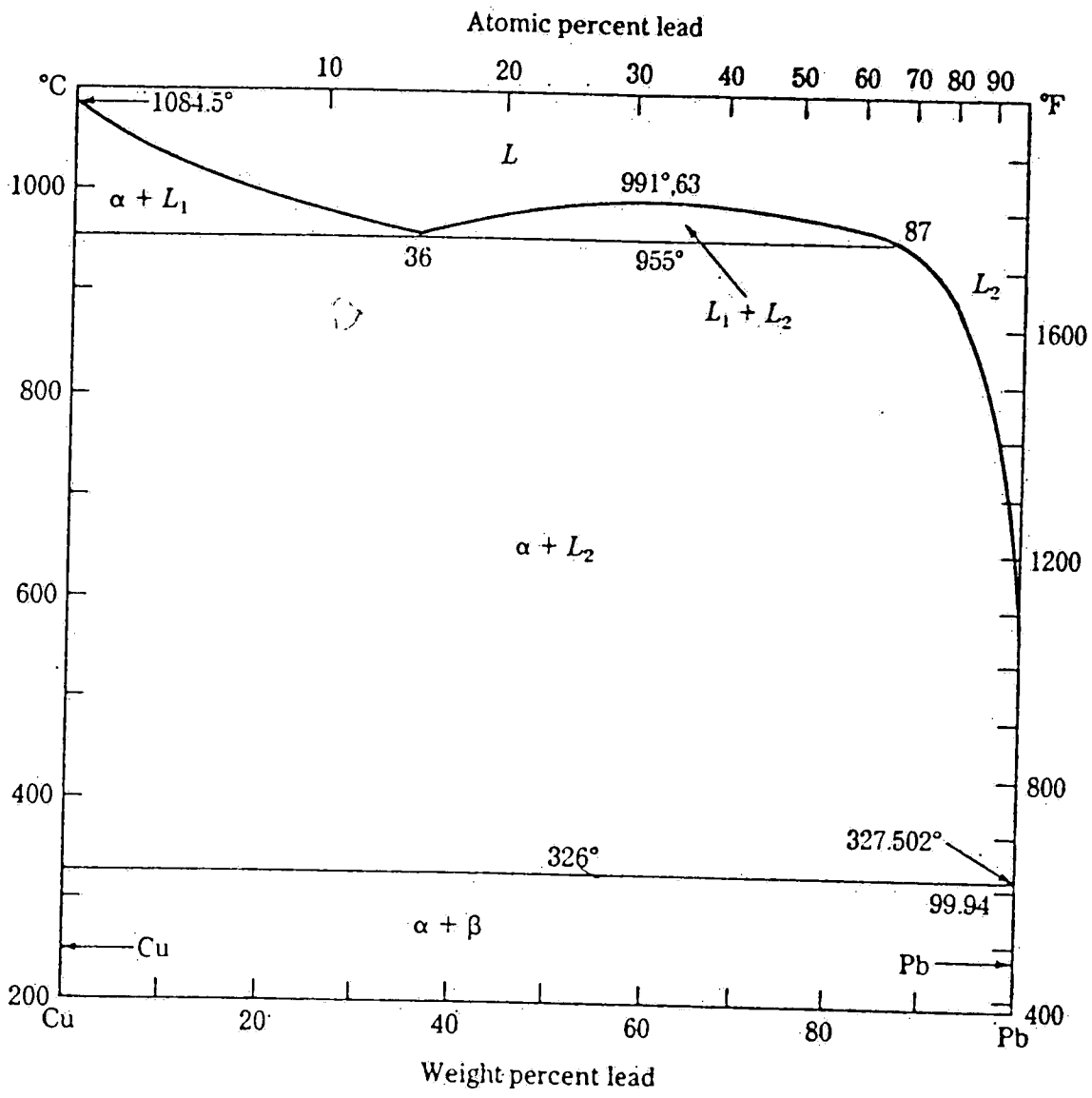

FIGURE 5 The copper-lead phase diagram

2.3. Ternary Systems

Ternary alloy systems are much more complicated and harder to understand than unary and binary systems. However, they are very interesting in the following sense: In the production of integrated circuits, we usually begin with binary systems. However, a binary alloy can dissolve from the substance with which it is in contact. We practically have a ternary system at the junction of the binary alloy and the bonding surface; therefore, we would like to know about ternary systems in order to be able to predict how they react to, for instance, the changes in temperature. From this point

of view, quaternary systems are also very interesting because at the junction of two binary systems we might have one of them dissolved in the other, leading to a quaternary system.

3. Description of Previous Work

There are many ways of storing a picture in the computer. Raster image and vector image methods are just two of them. With the first method, a line, for instance, is stored as a series of consecutive pixels on the screen; in other words, it is stored as a bitmap. With the second method, a line is identified and stored with its beginning and ending points. Therefore, the former takes much more space in memory than the latter does.

The raster image method uses up hundreds of kilobytes in memory to store a picture because it needs 300 bits to store a one-inch-length digitized line. As an example, let's calculate how much space is needed to store a picture on a sheet of paper of size 8.5 inches by 11 inches. The amount of space needed is

8.5*300*11*300 bits

which is approximately 1M bytes. This much information in a file is hard to handle, read, and therefore display on the screen because of its large size.

When displaying bitmaps on the screen, some resolution is eventually lost, compared to the original digitized image. Suppose we have a 640*480 monochrome VGA screen. This monitor allows us to display at most

640 * 480 = 307200 pixels

on the screen. Therefore, it is impossible to display all the

information contained in a bitmap and it is worthwhile to try to compress a bitmap file before storing it in the computer. We can use the words "display" and "store" in the same context when talking about bitmaps because it is extremely difficult to extract useful information from a bitmap. Hence, a bitmap file can only be used for displaying purposes.

One way of compressing a bitmap file is to scan it 8*8 bits at a time and replace 8*8 bits of information with only one bit. A simple check suffices to do likewise; if any pixel among the read-in 8*8 bits is on, we have a 1 in the reduced image; otherwise a 0. Therefore, with this technique, 8*8 bits on the original image correspond to 1 bit on the reduced image. In other words, if this kind of compression is used, the size of the reduced image is one sixty-fourth of the original image. This transformation, as stated above, is unidirectional since it is impossible to get the original image back from the reduced image.

Earlier, we calculated that we needed approximately 1M bytes to store an 8.5 inches by 11 inches page. With the compression explained above, we need only

1M bytes / 64 = 15.7K bytes

and this is quite a reduction in size with some loss in resolution.

If we want to display a bitmap image on the screen, a small program, which reads in the image file and draws the corresponding picture on the screen, can be written.

After a little information about bitmaps and how compression can be made, we can now turn to the previous work done on the phase diagrams. Prior to the current work, a phase diagram to be displayed on the screen was digitized using an image scanner which can save the scanned image in different formats, such as pcx, binary, etc. After the image was scanned and saved in bitmap format, it was compressed using the technique explained above. The size of the reduced image file was around 12K bytes. Then, another program, written to display a bitmap image on the screen, was called to do so.

3.1 Problems with Previous Work

Although the previous work done on phase diagrams succeeded in displaying them on the screen with some loss in resolution, there are some problems with that work. These problems originate from using bitmaps, in general, for the representation of phase diagrams. Let's take a look at these problems.

The main problem is that it is extremely difficult to retrieve useful information from a bitmap representation. As explained in the previous section, after storing a picture in binary format in a file, all the information this file contains is pixels and their color attributes, gray scale values or just on or off, associated with them. Looking at such a file, we have no notion of what the information means. We cannot directly see what a CGM (Computer Graphics

Metafile), CDR (CorelDraw), or a PCX (PC Paintbrush) file has in it, either. However, these files can be easily decoded to see what is in them since we know how they are encoded. Since we want to make inferences on phase diagrams, we should not employ a representation using a binary bitmap format to encode pictures.

In order to enter a picture in the computer for storage in any format, it is first digitized using an image scanner. When a picture is scanned, there will be some loss in resolution because the scanner cannot capture everything in the picture. Meanwhile, compression of a file in binary format using the technique explained in the previous section, too, contributes to the loss in resolution. Therefore, some text such as the labels on the axes of phase diagrams are very hard to read on the screen when a file containing pictures in binary format is displayed.

The third problem, originating from using a binary format, as explained in the previous paragraph, is that we do not have a nice graphical interface. This is simply because of the loss in resolution after digitizing a picture and then compressing it.

Another problem is the amount of space a scanned image takes in memory. Just after digitizing a picture, it requires approximately 800K bytes. Let's suppose we have 10 phase diagrams to digitize. In this case, we need 8M bytes to store these image files. After the compression is used, we need

about 12.5K bytes per image file. The total space needed for 10 phase diagrams is more than 120K bytes. Even after the compression, we need a lot of space in memory to store phase diagrams.

In our work, using another representation scheme, we try to solve the problems stated above.

4. Compact Representation of Phase Diagrams

As expressed in the earlier sections, previous work has shown several problems in dealing with phase diagrams. In order to tackle these problems, we need to develop a different approach to represent phase diagrams, which must be easy to decode, easy to retrieve information from, should not take too much memory for storage, also we should have a nice graphical interface.

In the software market, there are some nice graphics tools, which are both powerful and easy to use, one of which is CorelDraw. CorelDraw has very powerful features, such as snapping a text to any curve, dragging characters in text until seeing the exact spacing, drawing curves quickly and easily, and editing curves. It also allows one to trace a picture.

Our approach to the compact representation of phase diagrams begins by first digitizing, as usual, a phase diagram using an image scanner. After tailoring the scanned image on the screen for our needs, we can save the resulting image. The scanner we used offers several formats in which an image can be saved. We chose to save the image in the PCX (PC Paintbrush) bitmap format which describes a graphic as a rectangle of black and white dots. The advantage of using the PCX format is that the files can be relatively small in many cases, 180K bytes as opposed to 800K bytes in binary format. On the other hand, a disadvantage of using the PCX format is

that we might need to be concerned with the resolution of a PCX file. Another disadvantage is that exported PCX files in CorelDraw contain no color or gray-scale information, everything is black or white. However, we do not have to worry about these details because we will be only tracing the diagrams saved in the PCX format.

Having saved a phase diagram in the PCX format, we can now trace it in order to save it in a different format, using CorelDraw. It is possible to import a file into CorelDraw and then export it. Importing a file into CorelDraw means reading in the file which is in a format other than CDR (CorelDraw). Likewise, exporting a file means saving it using a file format other than CDR. After importing a PCX file containing a phase diagram into CorelDraw, we trace the diagram, creating new lines, and curves when necessary, for each line or curve in the PCX file. After restoring all the information the PCX file contains, we remove the bitmap from the screen and save the remaining image in CGM (Computer Graphics Metafile) format.

Now we give some information about CGM format because it has been used again and again during this work.

4.1. Computer Graphics Metafile Format

The Computer Graphics Metafile provides a file format for the storage and retrieval of picture information. The file format consists of a set of elements that can be used to describe pictures in a way that is compatible between systems

of different architectures and devices of differing capabilities and design. The Computer Graphics Metafile allows picture information to be stored in an organized way on a graphical software system, and facilitates transfer of picture information between different graphical software systems and different computer graphics installations.

Every metafile starts with a BEGIN METAFILE element and ends with an END METAFILE element. This allows multiple metafiles to be stored or transferred together. Each picture starts with a BEGIN PICTURE element and ends with an END PICTURE element. Between these delimiters, the picture descriptor is separated from the picture body by a BEGIN PICTURE BODY element.

Binary encoding and character encoding are the ways of encoding a Computer Graphics Metafile. The binary encoding of CGM provides a representation of the metafile syntax that can be optimized for speed of generation and interpretation, while still providing a standard means of interchange among computer systems. The encoding uses binary data formats that are much more similar to the data representations used within computer systems than the data formats of the other encodings. Furthermore, some of the data formats may exactly match those of some computer systems.

All elements in the metafile are encoded using a uniform scheme. The elements are represented as variable length data structures, each consisting of opcode information (element

class plus element identifier), the length of its parameter data, and finally the parameter data, if any.

The structure of the Metafile (MF) is as follows:

BEGIN MF	MD	<picture>...	END MF

The BEGIN METAFILE element is followed by the METAFILE DESCRIPTOR (MD) which precedes the pictures. Finally, the Metafile is ended with an END METAFILE element.

The metafile is partitioned into pictures. A picture consists of a BEGIN PICTURE element, a PICTURE DESCRIPTOR element (PD), a BEGIN PICTURE BODY element, an arbitrary number of control, graphical, and attribute elements, and finally an END PICTURE element.

BEGIN PIC	PD	BEGIN BODY	<element>...	END PIC

The binary encoding of the metafile consists of a sequential collection of bits. For measuring the lengths of elements, the metafile is partitioned into octets, which are 8-bit fields. The structure is also partitioned into 16-bit fields called words. Metafile elements are constrained to start on word boundaries within the binary data structure.

Metafile elements are represented in one of two forms: short-form commands (binary-encoded elements) and long-form commands. There are two differences between them. The first one is that a short-form command always contains a complete element; the long-form command can accommodate partial

elements. The second difference is that a short-form command only accommodates parameter lists up to 30 octets in length; on the other hand, a long-form command can accommodate lengths up to 32767 octets per data partition.

Metafile elements are grouped into classes. Each element in a class has its own identifier to make the element unique. Each command, a binary-encoded element, has a command header. For short-form commands, the command header consists of a single word divided into three fields: element class, element id, and parameter list length, as shown in figure 6.

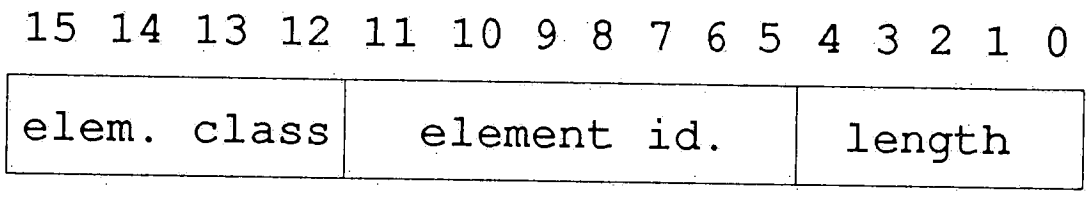

FIGURE 6 format of a short-form command header

In figure 6, bits 15 to 12 correspond to element class whose range is 0 to 15, whereas bits 11 to 5 and 4 to 0 correspond to element identifier (value range is 0 to 127) and parameter list length (value range is 0 to 30) respectively.

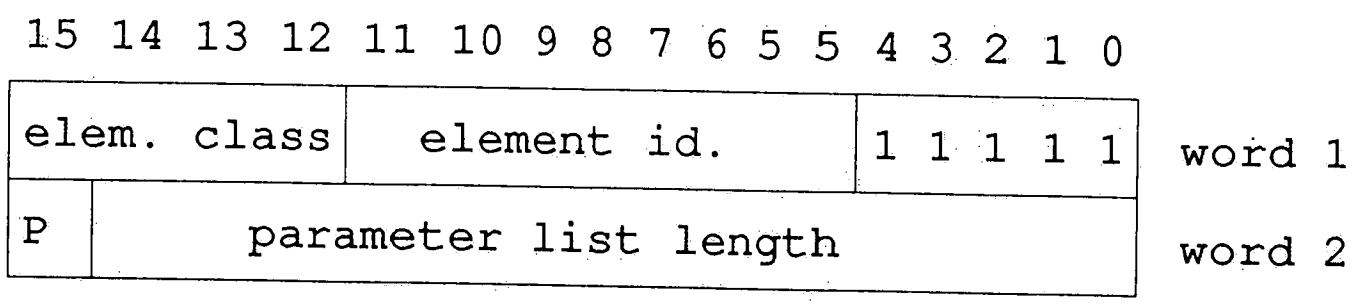

FIGURE 7 format of a long-form command header

If a command's parameter list length is graeter than 30,

the long-form command header is used instead of the short-form command header. The long-form command header is shown in figure 7, where P is the partition flag, whose value is 0 for last partition and 1 for non-last partition. Bits 4 to 0 in word 1, which is decimal 31, indicates that this is a long-form command header.

The most important metafile elements are VDC TYPE, INTEGER PRECISION, REAL PRECISION, VDC INTEGER PRECISION, VDC REAL PRECISION, POLYLINE, DISJOINT POLYLINE, TEXT, POLYGON, RECTANGLE, CIRCLE, ELLIPSE, TEXT PATH, and TEXT ALIGNMENT. VDC TYPE indicates the data type, integer or real, of the Virtual Device Coordinates (VDC). INTEGER PRECISION specifies the length, 8-bit, 16-bit, 24-bit, or 32-bit, for the operands of integer data types. VDC INTEGER PRECISION, like INTEGER PRECISION, specifies the length for the operands of data type point (P) and operands of data type VDC value; however, 8-bit is not permitted as precision. POLYLINE specifies a series of connected lines; a line is drawn from the first point in the parameter list to the second point, from the second point to the next point, etc., and from the next-to-last point to the last point. Character strings can be represented in CGM using TEXT with three parameters: a point at which the string begins, a flag which indicates whether the text string is final or not, and finally the string itself. POLYGON is similar to POLYLINE except it is assumed that after the last point in the parameter list comes the first point in the

parameter list.

4.2. Programs Working on CGM Files

Before writing a program to read from a CGM file and to display what is in this file on the screen, we should worry about adding some information to this file. After a phase diagram has been saved as a CGM file in CorelDraw, this file contains no information concerning the x and y axes of the diagram. Along the x axis, atomic percentages of the substances should be displayed, whereas along the y axis, the temperature should be displayed. In order for this information to be appended to the CGM file containing the phase diagram, a small program must be written. This program gets the information to be appended from the keyboard, encodes it so that it does not take too many bytes, and adds the encoded information to the end of the CGM file.

Sometimes we want to put some labels such as "liquid" inside regions on a phase diagram. For this, we need two small programs to do some file format conversions. The first program reads a binary file and converts it into a hexadecimal text file. The second program does the opposite: converts a hexadecimal text file into a binary file. Having had these two programs available and known the encoding of CGM files, we can easily put labels anywhere inside CGM files. First, we decide where to put the label on the phase diagram. Then we have the text containing the label in the CGM format. The third step is

to convert the CGM file, which is binary, into a hexadecimal text file. The next step is to put the information in a proper place, knowing the constraint that commands begin on word boundaries. The last thing to do is to convert the modified hexadecimal text file back into a binary file.

At this point, we have all the information we need, labels, temperature, and atomic percentages, available to us in order for a phase diagram to be complete.

4.2.1. The program draw.c

Once we are done with a phase diagram, storing the diagram as a CGM file, putting labels inside regions, adding the information to be displayed along the axes of the diagram, we can call the executable version of draw.c to display the diagram on the screen. However, before doing this, we have to make sure that the grid to be displayed with the phase diagram must be saved as a CGM file and there must exist a file called egavga.bgi, which is needed when initializing the graphics, in the directory where draw.c is stored. The program draw.c as can be understood from its extension has been written in C.

Draw.c has the capability of decoding a CGM file and displaying what is contained in this file on the screen. It first reads the file grid.cgm and stores the information concerning the lines of the grid. Then it reads another CGM file supplied by the user and also stores the information contained in this file. Finally, it draws the diagram on the

screen and puts the labels on it, if any.

The most important data structures of draw.c are LineNode and TextNode; the former is used to maintain a singly-linked list of points, whereas the latter is used to store a character string with some attributes. LineNode and TextNode are used with two arrays, lineList and textList, respectively. LineList is an array with each element pointing to a LineNode; on the other hand, textList is an array with each element pointing to a TextNode. Each element of lineList either points to the first element of a linked list of points or is null. If an entry of lineList is not null, it points to a linked list which contains all points on a line. Therefore, each line on the diagram is represented in draw.c as a linked list of points pointed to by an element of lineList. Each element of textList either is null or points to a TextNode which contains a character string as well as its attributes such as text type, text path, and text alignment.

We assume that the x and y coordinates of a point are represented by integers, not by real numbers. If real numbers are used, we have to add several functions to the program, draw.c, which is actually not too complicated to do anyway.

After explaining what draw.c does, let's give some information about some of its functions. The most important functions in this program are draw_2D, GetInfo, and main. The function main consists of, in general, two similar loops. Inside each loop, a command is read from a CGM file, the

element class, element id, and parameter list length are extracted from the command, the command code is calculated from element class and element id, and finally the function GetInfo is called with two arguments, command code and paramater list length, to decide on the actions to take. The first loop is for the CGM file containing the grid, whereas the second loop is for the CGM file containing the phase diagram.

The function GetInfo can deal with 14 of the CGM commands, the ones that concern us most. GetInfo consists of a huge switch statement and checks if there is any error for each command. If no errors, it takes the necessary actions such as storing the line in a linked list, storing text information, or changing the defaults for metafile descriptors and control elements. The metafile commands that GetInfo can deal with are VDC TYPE, INTEGER PRECISION, REAL PRECISION, METAFILE DEFAULTS REPLACEMENT, VDC INTEGER PRECISION, VDC REAL PRECISION, POLYLINE, DISJOINT POLYLINE, TEXT, RESTRICTED TEXT, APPEND TEXT, POLYGON, POLYGON SET, RECTANGLE, TEXT PATH, and TEXT ALIGNMENT.

The function draw_2D simply draws lines and puts labels on the screen. Since elements of lineList point to first points of lines, draw_2D draws a line from the first point on the linked list to the second point, ..., from the next-to-last point on the linked list to the last point on the list. The text of each element of textList is displayed on the

screen with text path and horizontal and vertical alignments taken into consideration.

Some of the subsidiary functions in draw.c are GetWord, GetOctets, freeMemory, statusLine, drawBorder, mainWindow, and displayScale. The flowcharts corresponding to the functions main and GetInfo are shown in figures 8-a and 8-b, respectively.

The output of draw.c for the lead-tin phase diagram is shown in figure 9. The original lead-tin phase diagram taken from Bulletin of Alloy Phase Diagrams [6] is shown in figure 10 to make a comparison between the two diagrams.

4.3. Achievements

So far, we have a very compact representation of phase diagrams (less than 2K bytes in size per diagram) using Computer Graphics Metafile format. We also have a nice display of phase diagrams, with some labels on them, on the screen. Therefore, we have, up to this point, solved two of the problems the previous work faced.

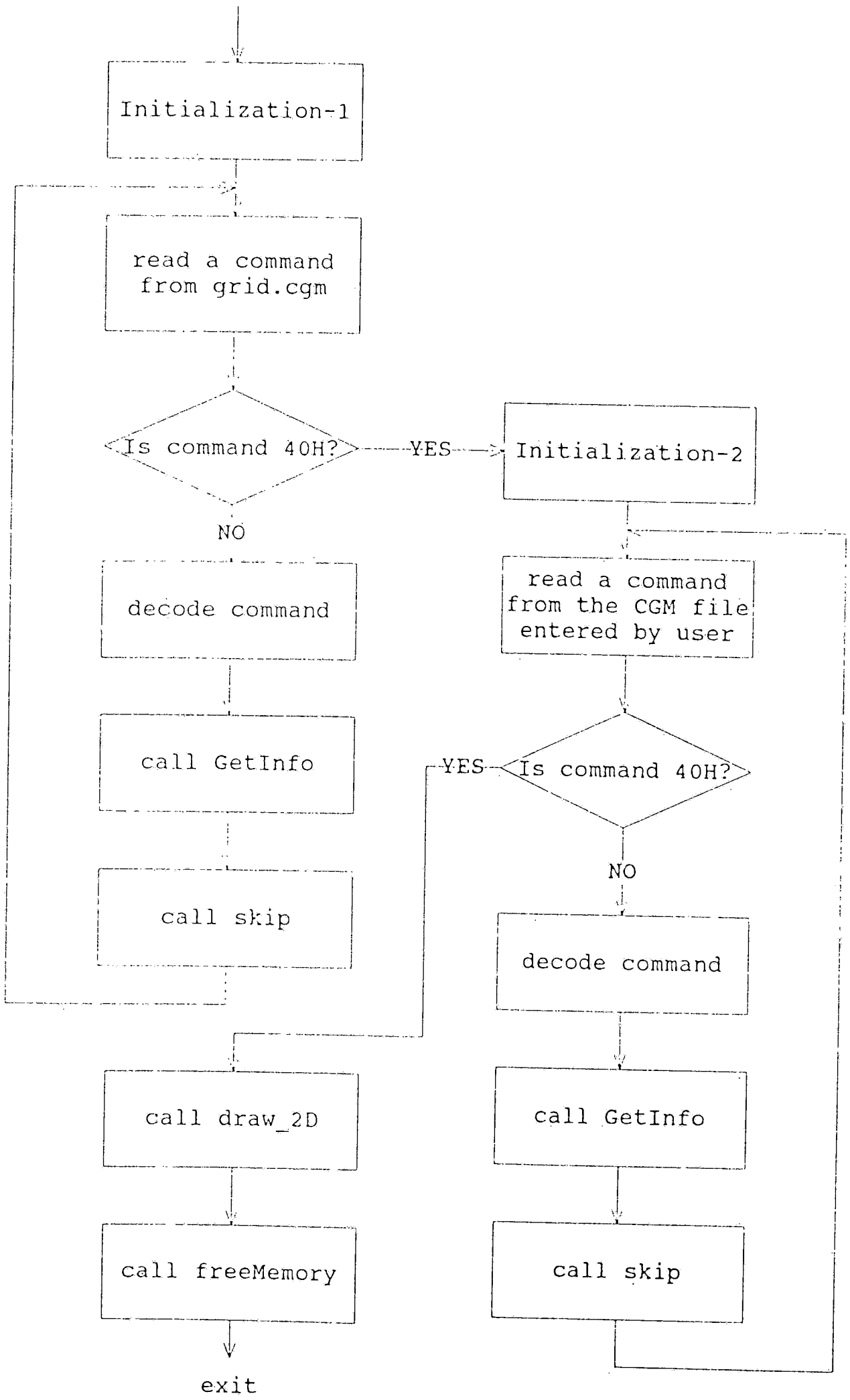

FIGURE 8-a flowchart of the function main

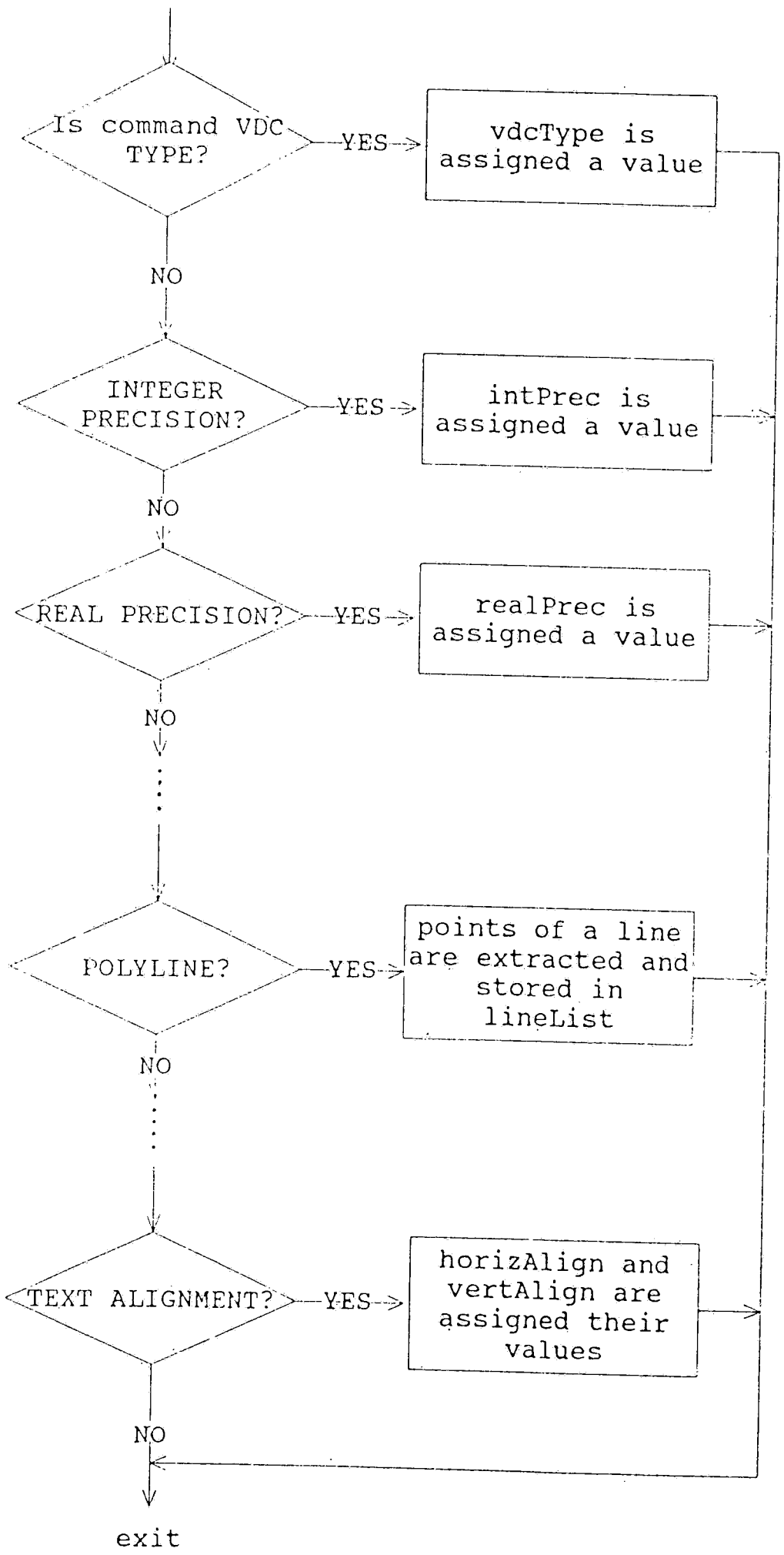

FIGURE 8-b flowchart of GetInfo

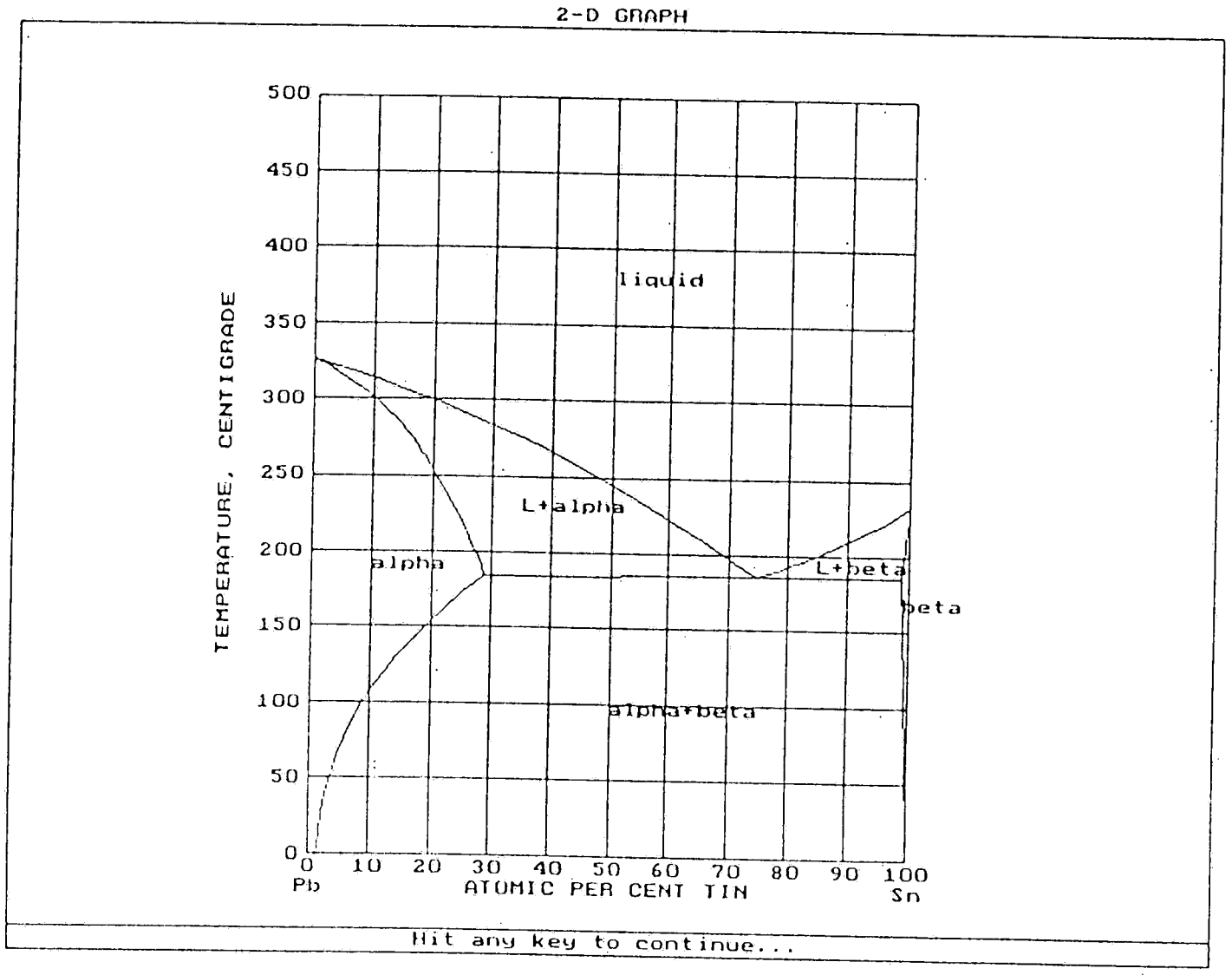

FIGURE 9 output of draw.c for the lead-tin phase diagram

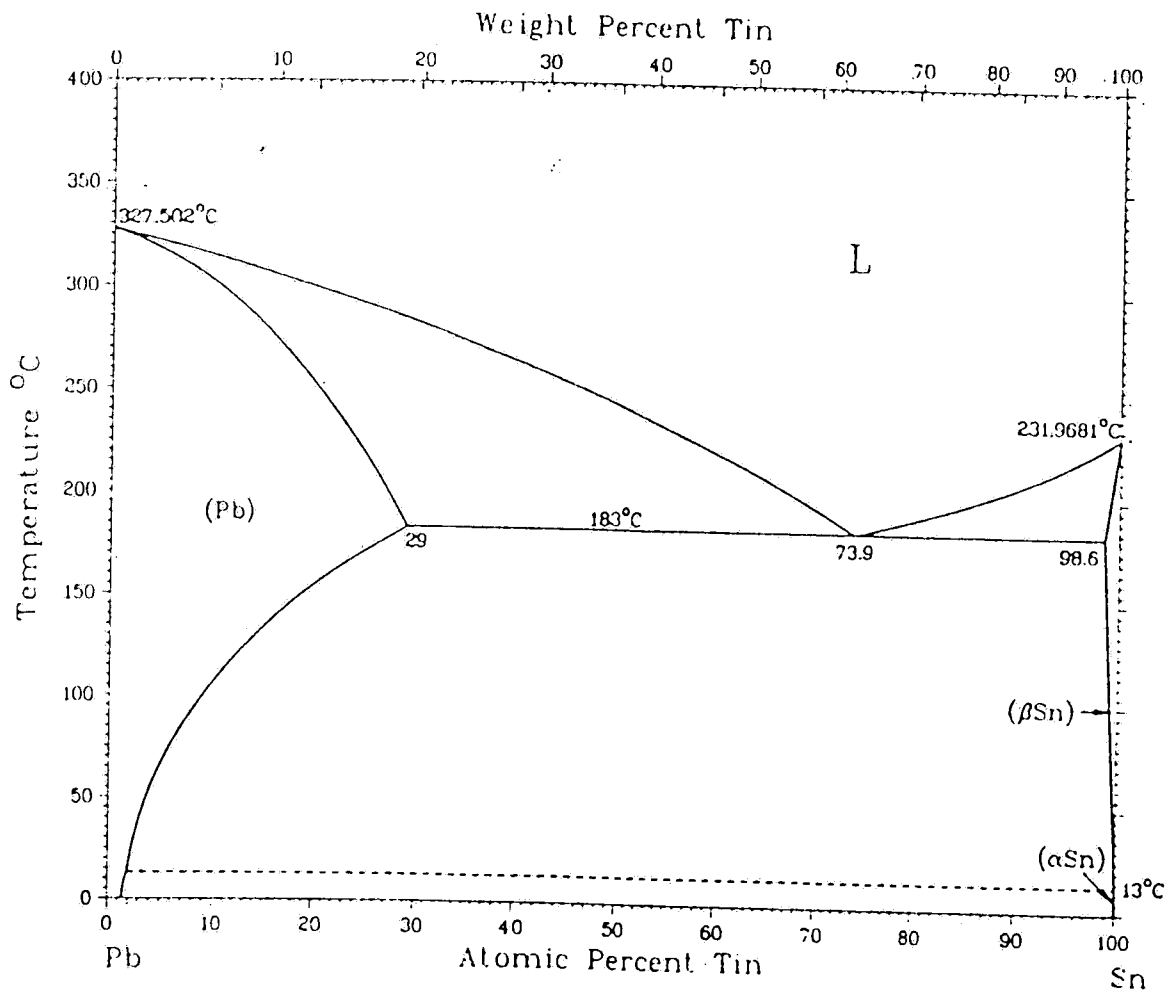

FIGURE 10 phase diagram for the lead-tin alloy, taken from Bulletin of Alloy Phase Diagrams [6]

5. Inference on Stored Phase Diagrams

The first part of the work we have done is to compactly represent phase diagrams in the computer and display them on the screen, as explained in the previous sections. The second part is to make some inferences on the stored phase diagrams. As expressed earlier, we need a representation scheme other than bitmap formats in order to retrieve and use the information contained in the diagrams. In this section, we will explain ways of making inferences from phase diagrams, such as identifying regions on the diagrams, calculating the lowest melting point, etc. We will also explain some CLIPS rules that were written to do so. CLIPS (C Language Integrated Production System) is a forward chaining rule-based language that has inferencing capabilities. Designed at NASA/Johnson Space Center with the specific purposes of providing high portability, low cost, and easy integration with external systems, CLIPS was written using the C programming language. CLIPS can be integrated with external C functions or applications and we will make use of this feature of CLIPS.

Before trying to perform inference on a phase diagram, all the information, such as lines and text, it contains must be asserted into the CLIPS fact base. Therefore, the CGM file containing the phase diagram we are trying to make inferences from must be read in first. CLIPS has two built-in functions, read and readline, to get information from the keyboard or a file. But these two functions are written to read ASCII text,

either from the keyboard or a file, not binary information. Since CGM files are in binary format and built-in CLIPS functions cannot deal with binary files, we have to write our own function to read from a CGM file. This function named extract_info reads a CGM file and asserts information concerning lines and text of the phase diagram directly into the CLIPS fact base. Extract_info is a modified version of draw.c in that after accumulating information for a line or text, it asserts that information into the CLIPS fact base instead of putting that information in one of the two arrays, lineList and textList, as draw.c does. Extract_info also does not have the display capabilities of draw.c.

In order to facilitate identification of regions on a phase diagram, each region to be identified is labeled with a name, such as liquid, liquid+alpha, or alpha+beta, depending on the characteristics of the region.

The collection of rules to make inferences from phase diagrams is named idreg.clp. Idreg.clp first calls the function extract_info to transfer the information concerning lines and text from a CGM file into the CLIPS fact base. The next step performed is to find intersection points of the lines and the rectangle enclosing the diagram. Then, the edges of the rectangle are split up into line segments based on the number of intersection points on that line. For example, if an edge has two intersection points on it, it is split up into three line segments. The next step is to find out which line

is the uppermost line of the region to be identified. The uppermost line of the region is asserted into the fact base as the first segment of the polygon corresponding to the region. After this point, each line segment which continues from the last point of the polygon and makes the smallest left (or right, if there are no lines with a left angle) angle gets appended to the polygon until the first and last points of the polygon are unified. This means that the polygon we have found corresponds to the region whose borders we are trying to determine and the edges of the polygon are the borders of the region.

Each region is identified in the counterclockwise direction. Unlike convex polygons, concave polygons are said to be hard to identify and there are not many algorithms to do so. Therefore, the author had to devise an algorithm which can be used to identify both kinds of polygons, convex and concave. If the region whose borders we are trying to find is convex at the current point, it makes a left turn from the last line segment of the polygon ([8]). In other words, the angle between the previous and next line segments of the polygon, corresponding to the region, is not reflexive. If the region is concave at the current point, it makes a right turn.

Let's explain this concept with the help of fig. 11. Let's suppose we are currently at point p2 and the coordinates of p1, p2, and p3 are (x1,y1), (x2,y2), and (x3,y3) respectively. Deciding whether angle (p1p2p3) is a left or

right turn corresponds to evaluating a 3*3 determinant in the points' coordinates. The determinant

$$\Delta = \begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix}$$

gives twice the signed area of the triangle (p1p2p3), where the sign of the determinant Δ is positive if and only if (p1p2p3) form a counterclockwise cycle. Therefore, the angle p1p2p3 is a left turn if and only if the determinant Δ is positive.

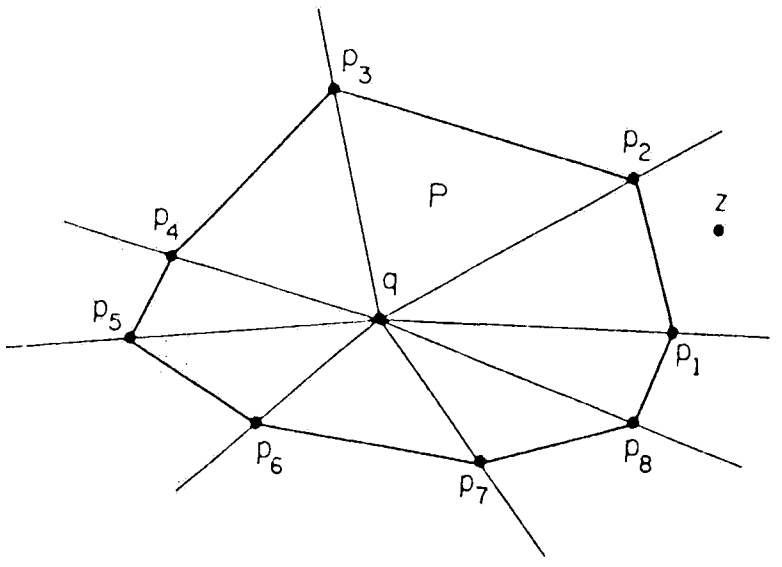

FIGURE 11 a polygon P with vertices labelled

In a phase diagram, if there is more than one line beginning or ending at any intersection point on the borders of a region, we calculate which lines make left turns and which ones make right turns. After finding out which lines make left turns and right turns, we further process only one kind of turn : if there are any lines making left turns, they get processed; otherwise, the lines making right turns get processed. For each line to be processed, we calculate the

angle it makes with the last line segment of the region at the current point. Then, we choose the line among others which makes the smallest angle with the last line segment of the region. This way, we are guaranteed to find the correct polygon for the region whose borders we are trying to determine.

The listing of idreg.clp is given in the Appendix. The output produced by idreg.clp for the lead-tin phase diagram is shown in fig. 12. The information concerning format of the facts of fig.12 is given at the very beginning of idreg.clp in the Appendix.

```
(lowest-x 0)
(lowest-y 0)
(highest-x 1000)
(highest-y 1000)
(line 1 1000   0 998  11 998  23 997  46 997  69 996  92
        995 115 995 138 994 161 993 184 992 207 991 230
        990 253 989 276 988 300 987 323 986 346 985 370)
(line 2  985 370 985 373 986 376 987 383 988 389  989 395
        989 400 990 406 991 411 992 416 993 422  994 427
        995 433 996 438 996 444 997 450 998 456 1000 463)
(line 3 1000 463 988 458 979 454 960 445 942 437 926 429
        910 423 895 417 881 411 866 406 853 401 838 397
        824 392 810 388 795 384 779 379 762 375 745 370)
(line 4 745 370 722 382 699 394 654 416 610 438 566 459
        522 479 479 498 435 516 391 533 346 550 301 566
        254 582 207 596 157 611 107 625  54 638   0 652)
(line 5   0 652   7 649  14 646  28 641  42 635  55 628
         68 622  80 615  92 608 103 601 113 593 124 585
        134 577 144 568 153 559 162 550 172 540 181 530
        184 526 188 521 195 513 202 504 209 495 216 486
        223 477 230 467 237 458 244 448 251 438 258 427
        265 416 271 405 278 393 284 381 290 368)
(line 6 290 368 272 357 255 346 224 324 195 303 168 282
        145 260 123 239 104 217  87 196  72 173  59 151
         48 127  38 103  30  78  24  53  19  26  15   0)
```

FIGURE 12 output of idreg.clp for the lead-tin system

```
(line 7 290 368 745 370)
(line 8 745 370 985 370)
(min-temperature 0)
(advance-in-temp 50)
(line 9      0 1000 1000 1000)
(line 10 1000 1000 1000  463)
(line 11 1000  463 1000    0)
(line 12 1000    0   15    0)
(line 13   15    0    0    0)
(line 14    0    0    0  652)
(line 15    0  652    0 1000)
(max-line-number 15)
(borders 9 15 4 3 10 for liquid)
(polygon 1000 1000    0 1000   0 652  54 638 107 625 157 611
          207  596  254  582 301 566 346 550 391 533 435 516
          479  498  522  479 566 459 610 438 654 416 699 394
          722  382  745  370 762 375 779 379 795 384 810 388
          824  392  838  397 853 401 866 406 881 411 895 417
          910  423  926  429 942 437 960 445 979 454 988 458
         1000  463 1000 1000 for liquid)
(min-melting-temp 185)
(borders 4 5 7 for L+alpha)
(polygon 745 370 722 382 699 394 654 416 610 438 566 459
          522 479 479 498 435 516 391 533 346 550 301 566
          254 582 207 596 157 611 107 625  54 638   0 652
```

(figure 12, continued from last page)

7 649 14 646 28 641 42 635 55 628 68 622

80 615 92 608 103 601 113 593 124 585 134 577

144 568 153 559 162 550 172 540 181 530 184 526

188 521 195 513 202 504 209 495 216 486 223 477

230 467 237 458 244 448 251 438 258 427 265 416

271 405 278 393 284 381 290 368 745 370

for L+alpha)

(borders 3 8 2 for L+beta)

(polygon 1000 463 988 458 979 454 960 445 942 437 926 429

910 423 895 417 881 411 866 406 853 401 838 397

824 392 810 388 795 384 779 379 762 375 745 370

985 370 985 373 986 376 987 383 988 389 989 395

989 400 990 406 991 411 992 416 993 422 994 427

995 433 996 438 996 444 997 450 998 456 1000 463

for L+beta)

(borders 2 1 11 for beta)

(polygon 1000 463 998 456 997 450 996 444 996 438 995 433

994 427 993 422 992 416 991 411 990 406 989 400

989 395 988 389 987 383 986 376 985 373 985 370

986 346 987 323 988 300 989 276 990 253 991 230

992 207 993 184 994 161 995 138 995 115 996 92

997 69 997 46 998 23 998 11 1000 0 1000 463

for beta)

(borders 5 14 13 6 for alpha)

(figure 12, continued from last page)

```
(polygon 290 368 284 381 278 393 271 405 265 416 258 427
         251 438 244 448 237 458 230 467 223 477 216 486
         209 495 202 504 195 513 188 521 184 526 181 530
         172 540 162 550 153 559 144 568 134 577 124 585
         113 593 103 601  92 608  80 615  68 622  55 628
          42 635  28 641  14 646   7 649   0 652   0   0
          15   0  19  26  24  53  30  78  38 103  48 127
          59 151  72 173  87 196 104 217 123 239 145 260
         168 282 195 303 224 324 255 346 272 357 290 368
         for alpha)
(borders 7 6 12 1 8 for alpha+beta)
(polygon 745 370  290 368 272 357 255 346 224 324 195 303
         168 282  145 260 123 239 104 217  87 196  72 173
          59 151   48 127  38 103  30  78  24  53  19  26
          15   0 1000   0 998  11 998  23 997  46 997  69
         996  92  995 115 995 138 994 161 993 184 992 207
         991 230  990 253 989 276 988 300 987 323 986 346
         985 370  745 370 for alpha+beta)
```

(figure 12, continued from last page)

6. Conclusions and Suggestions for Future Work

Conclusions

Many industrial operations depend on knowledge of phase diagrams which are graphical representations of phase changes involving one or more substances. The electronics industry is just one example. Compositions of two or more metals, called the alloys, are mainly used in semiconductor electronics in assembly and attaching something to integrated circuit packages. Since alloys are broadly used, it is important that we know their characteristics and how to store them in computers. Phase diagrams can be stored using many available file formats. In this work, we have explained two ways of doing so, the binary bitmap format and Computer Graphics Metafile format, and have chosen the latter. In the first part of the work, we succeeded in obtaining a very compact representation of phase diagrams, less than 2K bytes in memory, and a very nice graphics interface. When a phase diagram is displayed on the screen using the programs mentioned earlier, everything on the diagram is readable, as opposed to not being able to read most of the labels on the diagram with the binary bitmap format previously used for representation of phase diagrams.

Another issue that was addressed is the problem of retrieving information from phase diagrams. This is almost impossible with the binary bitmap formats. Since Computer

Graphics Metafile format uses lines to represent pictures, generally, and we know how to decode CGM files, we can easily retrieve the information we want from phase diagrams. CLIPS rules were written, by the author, to identify regions on the phase diagrams. This is quite useful in doing some calculations, for instance the lowest melting temperature. We have succeeded in finding out all the regions on phase diagrams of interest.

Overall, we have a very compact representation of phase diagrams, can display them very easily on the screen, and most importantly we can make inferences on stored phase diagrams, using the Computer Graphics Metafile format.

Suggestions for Future Work

In this work, we have concentrated on binary phase diagrams. Both parts of our work can be extended to include ternary and quaternary phase diagrams. In the first part, the drawing program, draw.c, can be extended to read three-dimensional phase diagrams and to display them on a two-dimensional screen. In the second part, more rules can be added to the rules, idreg.clp, to find volumes of ternary phase diagrams as well. However, these two extensions might require a lot of effort because the current representation scheme of phase diagrams, Computer Graphics Metafile, has to be changed since it cannot represent three-dimensional pictures.

Another aspect of extending the current work could be adding a hierarchical representation to the CLIPS rules we have, which can be done using CLIPS 5.0 which supports object-oriented frame structures. Using this kind of representation, we can derive edges of a surface from vertices, surfaces from edges, and finally volumes from surfaces.

7. Bibliography

[1] Orser, Don J., "An Algebraic Representation for the Topology of Multicomponent Phase Diagrams", Center for Manufacturing Engineering, National Bureau of Standards, 1986

[2] Alper, Allen M., "Phase Diagrams: Materials Science and Technology", volume 1, Academic Press, New York, 1970

[3] Smith, William F., "Principles of Materials Science and Engineering", 2nd edition, McGraw-Hill, New York, 1990

[4] "CorelDraw", Corel Systems Corporation, 1988

[5] "Computer Graphics Metafile", American National Standards Institute, New York, 1987

[6] "Bulletin of Alloy Phase Diagrams", American Society for Metals, Metals Park, Ohio

[7] Giarratano, Joseph C. and Riley, Gary D., "Expert Systems: Principles and Programming", PWS-KENT Publishing Company, Boston, 1989

[8] Preparata, Franco P. and Shamos, Michael I., "Computational Geometry : An Introduction", Springer-Verlag, New York, 1985

[9] "CLIPS Reference Manual, Version 4.3 of CLIPS", Artificial Intelligence Section, Lyndon B. Johnson Space Center, July 1989

[10] "Clips Reference Manual", version 5.0, Software Technology Branch, Lyndon B. Johnson Space Center, volumes 1 and 2, January 1991

8. Appendix

Listing of idreg.clp which is a compilation of rules to make inferences on stored phase diagrams is given below.

```
; idreg.clp
; 4/25/91
;
; fact templates used in the rules below
;     (text <text-number> <string> at <x> <y>)
;     (special-line <x1> <y1> <x2> <y2> for <phase>)
;     (closest-point <x> <y> on <line-number> for <phase>)
;     (line <line-number> <<< <x> <y> >>>)
;     (intersects-with <line-number> at <x> <y> for <phase>)
;     (t-poly <<< <x> <y> >>> for <phase>)
;     (intersection-points << <x> <y> >> on <line-number>)
;     (IPs << <x> <y> >> on <line-number>)
;     (polygon <<< <x> <y> >>> for <phase>)
;     (temp <line-number> for <phase>)
;     (angle      <angle> to <line-number> for <phase>)
;     (min-angle <angle> to <line-number> for <phase>)
;     (has-same-angle  <<<line-number>>> for ?phase)
;     (process <<< <x> <y> >>> for ?phase)
;     (point-no <number> for <phase>)
;     (left-turn  <<<line-number>>> for ?phase)
;     (right-turn <<<line-number>>> for ?phase)
```

```
;     (append-line <line-number> for <phase>)
;     (borders <<<line-number>>> for <phase>)
;     (min-temperature <minTemp>)
;     (advanca-in-temp <advance>)
;     (min-melting-temp <temperature>)
;     (max-line-number <max>)
;     (lowest-x <x>)
;     (lowest-y <y>)
;     (highest-x <x>)
;     (highest-y <y>)

(defrule readFile
  ?init <- (initial-fact)
  =>
  (retract ?init)
  (printout t "Input File Name : ")
  (bind ?file-name (str-cat (readline) ".cgm"))
  (extract_info ?file-name)
  (assert (IPs on -1)
          (IPs on -2)
          (IPs on -3)
          (IPs on -4)))

(defrule findLiquid
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member liquid $?txt) 0))
```

```
  =>
  (retract ?tt)
  (assert (special-line ?x ?y ?x 2000 for liquid)
          (closest-point 5000 5000 on 0 for liquid)))

(defrule findL+alpha
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member L+alpha $?txt) 0))
  =>
  (retract ?tt)
  (assert (special-line ?x ?y ?x 2000 for L+alpha)
          (closest-point 5000 5000 on 0 for L+alpha)))

(defrule findL+beta
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member L+beta $?txt) 0))
  =>
  (retract ?tt)
  (assert (special-line ?x ?y ?x 2000 for L+beta)
          (closest-point 5000 5000 on 0 for L+beta)))

(defrule findBeta
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member beta $?txt) 0))
  =>
  (retract ?tt)
```

```
    (assert (special-line ?x ?y ?x 2000 for beta)
            (closest-point 5000 5000 on 0 for beta)))

(defrule findAlpha
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member alpha $?txt) 0))
  =>
   (retract ?tt)
   (assert (special-line ?x ?y ?x 2000 for alpha)
           (closest-point 5000 5000 on 0 for alpha)))

(defrule findAlpha+Beta
  ?tt <- (text ?tno $?txt at ?x ?y)
  (test (!= (member alpha+beta $?txt) 0))
  =>
  (retract ?tt)
  (assert (special-line ?x ?y ?x 2000 for alpha+beta)
          (closest-point 5000 5000 on 0 for alpha+beta)))

(defrule findIPs
  (declare (salience 30))
  (line ?lno1&:(< ?lno1 0) ?xb ?yb ?xe ?ye)
  ?ip <- (IPs $?pts1 on ?lno1)
  (line ?lno2&:(> ?lno2 0) ?x1 ?y1 $?pts2 ?x2 ?y2)
  (test (and (or (!= ?x1 ?xb) (!= ?y1 ?yb))
             (or (!= ?x1 ?xe) (!= ?y1 ?ye))
```

```
                (or (!= ?x2 ?xb) (!= ?y2 ?yb))
                (or (!= ?x2 ?xe) (!= ?y2 ?ye)))))
=>
(bind ?len (length $?pts1))
(bind ?found1 0)
(bind ?found2 0)
(bind ?i 0)
(while (and (< ?i ?len) (< ?found1 1)) do
  (bind ?xnth (nth (+ ?i 1) $?pts1))
  (bind ?ynth (nth (+ ?i 2) $?pts1))
  (if (and (= ?x1 ?xnth) (= ?y1 ?ynth)) then
    (bind ?found1 1)
   else
    (bind ?i (+ ?i 2))))
(bind ?i 0)
(while (and (< ?i ?len) (< ?found2 1)) do
  (bind ?xnth (nth (+ ?i 1) $?pts1))
  (bind ?ynth (nth (+ ?i 2) $?pts1))
  (if (and (= ?x2 ?xnth) (= ?y2 ?ynth)) then
    (bind ?found2 1)
   else
    (bind ?i (+ ?i 2))))
(bind $?stuff $?pts1)
(bind ?remove 0)
(if (= ?yb ?ye) then
  (if (and (< ?found1 1)
```

```
            (= ?y1 ?yb)
            (or (and (>= ?x1 ?xb) (<= ?x1 ?xe))
                (and (>= ?x1 ?xe) (<= ?x1 ?xb)))) then
  (bind $?stuff (mv-append $?stuff ?x1 ?yb))
  (bind ?remove 1)
else (if (and (< ?found2 1)
                (= ?y2 ?yb)
                (or (and (>= ?x2 ?xb) (<= ?x2 ?xe))
                    (and (>= ?x2 ?xe) (<= ?x2 ?xb)))) then
    (bind $?stuff (mv-append $?stuff ?x2 ?yb))
    (bind ?remove 1)))
else   ; if (= ?xb ?xe) then
  (if (and (< ?found1 1)
           (= ?x1 ?xb)
           (or (and (>= ?y1 ?yb) (<= ?y1 ?ye))
               (and (>= ?y1 ?ye) (<= ?y1 ?yb)))) then
    (bind $?stuff (mv-append $?stuff ?xb ?y1))
    (bind ?remove 1)
  else (if (and (< ?found2 1)
                (= ?x2 ?xb)
                (or (and (>= ?y2 ?yb) (<= ?y2 ?ye))
                    (and (>= ?y2 ?ye) (<= ?y2 ?yb)))) then
    (bind $?stuff (mv-append $?stuff ?xb ?y2))
    (bind ?remove 1))))
(if (> ?remove 0) then
  (retract ?ip)
```

```
      (assert (IPs $?stuff on ?lno1))))

(defrule sortIPs
  "should fire after all findIPs rules on agenda fire"
  (declare (salience 20))
  (line ?lno&:(< ?lno 0) ?xb ?yb ?xe ?ye)
  ?ip <- (IPs $?pts on ?lno)
  =>
  (retract ?ip)
  (bind ?len (length $?pts))
  (if (= ?len 0) then
    (assert (intersection-points on ?lno))
  else
    (bind ?cpx ?xb)
    (bind ?cpy ?yb)
    (bind ?i 0)
    (bind $?stuff (str-explode ""))
    (while (< ?i ?len) do
      (bind ?j 0)
      (bind ?dist 5000)
      (while (< ?j ?len) do
        (if (= ?yb ?ye) then
          (if (> ?xe ?xb) then
            (bind ?d1 (- (nth (+ ?j 1) $?pts) ?cpx))
            (if (and (> ?d1 0) (< ?d1 ?dist)) then
              (bind ?dist ?d1)
```

```
      (bind ?tempx (nth (+ ?j 1) $?pts))
      (bind ?tempy (nth (+ ?j 2) $?pts)))
   else  ; if (> ?xb ?xe) then
    (bind ?d1 (- ?cpx (nth (+ ?j 1) $?pts)))
    (if (and (> ?d1 0) (< ?d1 ?dist)) then
      (bind ?dist ?d1)
      (bind ?tempx (nth (+ ?j 1) $?pts))
      (bind ?tempy (nth (+ ?j 2) $?pts))))
  else   ; if (= ?xb ?xe) then
   (if (> ?ye ?yb) then
     (bind ?d1 (- (nth (+ ?j 2) $?pts) ?cpy))
     (if (and (> ?d1 0) (< ?d1 ?dist)) then
       (bind ?dist ?d1)
       (bind ?tempx (nth (+ ?j 1) $?pts))
       (bind ?tempy (nth (+ ?j 2) $?pts)))
    else  ; if (> ?yb ?ye) then
     (bind ?d1 (- ?cpy (nth (+ ?j 2) $?pts)))
     (if (and (> ?d1 0) (< ?d1 ?dist)) then
       (bind ?dist ?d1)
       (bind ?tempx (nth (+ ?j 1) $?pts))
       (bind ?tempy (nth (+ ?j 2) $?pts)))))
   (bind ?j (+ ?j 2)))
(bind $?stuff (mv-append $?stuff ?tempx ?tempy))
(bind ?cpx ?tempx)
(bind ?cpy ?tempy)
(bind ?i (+ ?i 2)))
```

```
    (assert (intersection-points $?stuff on ?lno))))

(defrule splitLines
  (declare (salience 10))
  ?line <- (line ?lno&:(< ?lno 0) ?xb ?yb ?xe ?ye)
  ?ipt  <- (intersection-points $?pts on ?lno)
  ?mln  <- (max-line-number ?max)
  =>
  =>
  (retract ?line ?ipt ?mln)
  (bind ?len (length $?pts))
  (bind ?xf ?xb)
  (bind ?yf ?yb)
  (bind ?i 0)
  (while (< ?i ?len) do
    (bind ?max (+ ?max 1))
    (bind ?xl (nth (+ ?i 1) $?pts))
    (bind ?yl (nth (+ ?i 2) $?pts))
    (assert (line ?max ?xf ?yf ?xl ?yl))
    (bind ?xf ?xl)
    (bind ?yf ?yl)
    (bind ?i (+ ?i 2)))
  (bind ?max (+ ?max 1))
  (assert (line ?max ?xf ?yf ?xe ?ye)
          (max-line-number ?max)))

(defrule findIWs
```

```
(special-line ?x1 ?y1 ?x2 ?y2 for ?phase)
(line ?lno $?pts)
=>
(bind ?m1 2000000)
(bind ?len (length $?pts))
(bind ?i 2)
(while (< ?i ?len) do
  (bind ?x3 (nth (- ?i 1) $?pts))
  (bind ?y3 (nth ?i       $?pts))
  (bind ?x4 (nth (+ ?i 1) $?pts))
  (bind ?y4 (nth (+ ?i 2) $?pts))
  (if (!= ?x3 ?x4) then
    (bind ?m2 (/ (- ?y4 ?y3) (- ?x4 ?x3)))
   else
    (bind ?m2 2000000))
  (if (!= ?m1 ?m2) then
    (bind ?xi ?x1)
    (bind ?yi (- (+ (* ?m2 ?xi) ?y3) (* ?m2 ?x3)))
    (if (and
          (and (or (and (>= ?xi ?x1) (<= ?xi ?x2))
                   (and (>= ?xi ?x2) (<= ?xi ?x1)))
               (or (and (>= ?yi ?y1) (<= ?yi ?y2))
                   (and (>= ?yi ?y2) (<= ?yi ?y1))))
          (and (or (and (>= ?xi ?x3) (<= ?xi ?x4))
                   (and (>= ?xi ?x4) (<= ?xi ?x3)))
               (or (and (>= ?yi ?y3) (<= ?yi ?y4))
```

```
                        (and (>= ?yi ?y4) (<= ?yi ?y3))))) then
      (assert
           (intersects-with ?lno at ?xi ?yi for ?phase))))
    (bind ?i (+ ?i 2))))

(defrule findClosestPoint
  ?iw <- (intersects-with ?lno at ?xi ?yi for ?phase)
  ?cp <- (closest-point ?cpx ?cpy on ?cplno for ?phase)
  (special-line ?x1 ?y1 ?x2 ?y2 for ?phase)
  =>
  (retract ?iw)
  (bind ?sqcp (* (- ?y1 ?cpy) (- ?y1 ?cpy)))
  (bind ?sqi  (* (- ?y1 ?yi ) (- ?y1 ?yi )))
  (if (< ?sqi ?sqcp) then
    (retract ?cp)
    (assert (closest-point ?xi ?yi on ?lno for ?phase))))

(defrule assertInitialPoly
  (declare (salience -10))
  ?cp <- (closest-point $?pt on ?lno for ?phase)
  ?sl <- (special-line ?x1 ?y1 $?last-pt for ?phase)
  (line ?lno ?x2 ?y2 $?pts ?x3 ?y3)
  =>
  (retract ?cp ?sl)
  (bind ?delta
        (- (+ (* ?x1 ?y2) (* ?x2 ?y3) (* ?x3 ?y1))
```

```
                (+ (* ?y2 ?x3) (* ?y3 ?x1) (* ?y1 ?x2))))
    (if (> ?delta 0) then
      ; if the line is going from right to left
      (bind $?points (mv-append ?x2 ?y2 $?pts ?x3 ?y3))
     else
      ; if the line is going from left to right
      (bind $?points (mv-append ?x3 ?y3))
      (bind ?len (length $?pts))
      (while (> ?len 0) do
        (bind ?x (nth (- ?len 1) $?pts))
        (bind ?y (nth ?len       $?pts))
        (bind $?points (mv-append $?points ?x ?y))
        (bind ?len (- ?len 2)))
      (bind $?points (mv-append $?points ?x2 ?y2)))
    (assert (t-poly $?points for ?phase)
        (left-turn for ?phase)
        (right-turn for ?phase)
           (borders ?lno for ?phase)))

(defrule modifyLRTurns-1
  (declare (salience -20))
  (t-poly $?pts1 ?x1 ?y1 ?lpx ?lpy for ?phase)
  (line ?lno ?lpx ?lpy ?x2 ?y2 $?pts2)
  (borders $?lines for ?phase)
  (test (= (member ?lno $?lines) 0))
 ?rt <- (right-turn $?lineList-1 for ?phase)
```

```
?lt <- (left-turn  $?lineList-2 for ?phase)
(test (and (= (member ?lno $?lineList-1) 0)
           (= (member ?lno $?lineList-2) 0)))
=>
(bind ?delta
      (- (+ (* ?x1 ?lpy) (* ?lpx ?y2) (* ?x2 ?y1))
         (+ (* ?lpy ?x2) (* ?y2 ?x1) (* ?y1 ?lpx))))
(if (> ?delta 0) then    ; left-turn
 (retract ?lt)
 (assert (left-turn =(mv-append $?lineList-2 ?lno)
                    for ?phase))
else (if (< ?delta 0) then    ; right-turn
 (retract ?rt)
 (assert (right-turn =(mv-append $?lineList-1 ?lno)
                     for ?phase))
else  ; delta = 0
 (bind ?ax (- ?x1 ?lpx)) ; x component of vector a
 (bind ?ay (- ?y1 ?lpy)) ; y component of vector a
 (bind ?bx (- ?x2 ?lpx)) ; x component of vector b
 (bind ?by (- ?y2 ?lpy)) ; y component of vector b
 (bind ?a-len (sqrt (+ (* ?ax ?ax) (* ?ay ?ay))))
 (bind ?b-len (sqrt (+ (* ?bx ?bx) (* ?by ?by))))
 (bind ?a-mul-b (+ (* ?ax ?bx) (* ?ay ?by)))
 (bind ?angle (/ ?a-mul-b (* ?a-len ?b-len)))
 (bind ?angle (trunc (* ?angle 10000)))
 (if (> ?angle 10000) then  ; angle = 3.14159 radians
```

```
    (retract ?lt)
    (assert (left-turn =(mv-append $?lineList-2 ?lno)
                        for ?phase))
  else  ; angle = 0 radian
    (bind ?len2 0)
    (bind ?count 0)
    (while (and (= ?delta 0) (< ?count 5)) do
      (if (= ?len2 (length $?pts2)) then
        (bind ?x3 ?x2)
        (bind ?y3 ?y2)
       else
        (bind ?x3 (nth (+ ?len2 1) $?pts2))
        (bind ?y3 (nth (+ ?len2 2) $?pts2))
        (bind ?len2 (+ ?len2 2)))
      (bind ?delta
            (- (+ (* ?x1 ?lpy) (* ?lpx ?y3) (* ?x3 ?y1))
               (+ (* ?lpy ?x3) (* ?y3 ?x1) (* ?y1 ?lpx))))
      (bind ?count (+ ?count 1)))
  (bind ?count 0)
  (bind ?len1 (length $?pts1))
  (while (and (= ?delta 0) (< ?count 5)) do
    (if (= ?len1 0) then
      (bind ?x0 ?x1)
      (bind ?y0 ?y1)
     else
      (bind ?x0 (nth (- ?len1 1) $?pts1))
```

```
      (bind ?y0 (nth ?len1        $?pts1))
      (bind ?len1 (- ?len1 2)))
    (bind ?delta
          (- (+ (* ?x0 ?lpy) (* ?lpx ?y2) (* ?x2 ?y0))
             (+ (* ?lpy ?x2) (* ?y2 ?x0) (* ?y0 ?lpx))))
    (bind ?count (+ ?count 1)))
  (if (>= ?delta 0) then    ; left turn
    (retract ?lt)
     (assert (left-turn =(mv-append $?lineList-2 ?lno)
                         for ?phase))
    else  ; right turn
     (retract ?rt)
     (assert (right-turn =(mv-append $?lineList-1 ?lno)
                          for ?phase))))))

(defrule modifyLRTurns-2
  (declare (salience -20))
  (t-poly $?pts1 ?x1 ?y1 ?lpx ?lpy for ?phase)
  (line ?lno $?pts2 ?x2 ?y2 ?lpx ?lpy)
  (borders $?lines for ?phase)
  (test (= (member ?lno $?lines) 0))
  ?rt <- (right-turn $?lineList-1 for ?phase)
  ?lt <- (left-turn  $?lineList-2 for ?phase)
  (test (and (= (member ?lno $?lineList-1) 0)
             (= (member ?lno $?lineList-2) 0)))
  =>
```

```
(bind ?delta
      (- (+ (* ?x1 ?lpy) (* ?lpx ?y2) (* ?x2 ?y1))
         (+ (* ?lpy ?x2) (* ?y2 ?x1) (* ?y1 ?lpx))))
(if (> ?delta 0) then    ; left-turn
  (retract ?lt)
  (assert (left-turn =(mv-append $?lineList-2 ?lno)
                      for ?phase))
else (if (< ?delta 0) then    ; right-turn
  (retract ?rt)
  (assert (right-turn =(mv-append $?lineList-1 ?lno)
                       for ?phase))
else  ; delta = 0
  (bind ?ax (- ?x1 ?lpx)) ; x component of vector a
  (bind ?ay (- ?y1 ?lpy)) ; y component of vector a
  (bind ?bx (- ?x2 ?lpx)) ; x component of vector b
  (bind ?by (- ?y2 ?lpy)) ; y component of vector b
  (bind ?a-len (sqrt (+ (* ?ax ?ax) (* ?ay ?ay))))
  (bind ?b-len (sqrt (+ (* ?bx ?bx) (* ?by ?by))))
  (bind ?a-mul-b (+ (* ?ax ?bx) (* ?ay ?by)))
  (bind ?angle (/ ?a-mul-b (* ?a-len ?b-len)))
  (bind ?angle (trunc (* ?angle 10000)))
  (if (> ?angle 10000) then  ; angle = 3.14159 radians
    (retract ?lt)
    (assert (left-turn =(mv-append $?lineList-2 ?lno)
                        for ?phase))
  else  ; angle = 0 radian
```

```
(bind ?len2 (length $?pts2))
(bind ?count 0)
(while (and (= ?delta 0) (< ?count 5)) do
  (if (= ?len2 0) then
    (bind ?x3 ?x2)
    (bind ?y3 ?y2)
   else
    (bind ?x3 (nth (- ?len2 1) $?pts2))
    (bind ?y3 (nth ?len2       $?pts2))
    (bind ?len2 (- ?len2 2)))
  (bind ?delta
        (- (+ (* ?x1 ?lpy) (* ?lpx ?y3) (* ?x3 ?y1))
           (+ (* ?lpy ?x3) (* ?y3 ?x1) (* ?y1 ?lpx))))
  (bind ?count (+ ?count 1)))
(bind ?count 0)
(bind ?len1 (length $?pts1))
(while (and (= ?delta 0) (< ?count 5)) do
  (if (= ?len1 0) then
    (bind ?x0 ?x1)
    (bind ?y0 ?y1)
   else
    (bind ?x0 (nth (- ?len1 1) $?pts1))
    (bind ?y0 (nth ?len1       $?pts1))
    (bind ?len1 (- ?len1 2)))
(bind ?delta
        (- (+ (* ?x0 ?lpy) (* ?lpx ?y2) (* ?x2 ?y0))
```

```
                    (+ (* ?lpy ?x2) (* ?y2 ?x0) (* ?y0 ?lpx))))
  (bind ?count (+ ?count 1)))
(if (>= ?delta 0) then   ; left turn
  (retract ?lt)
  (assert (left-turn =(mv-append $?lineList-2 ?lno)
                      for ?phase))
 else  ; right turn
  (retract ?rt)
  (assert (right-turn =(mv-append $?lineList-1 ?lno)
                      for ?phase))))))

(defrule openUpLRTurns
  (declare (salience -30))
  ?lt <- (left-turn  $?lineList-1 for ?phase)
  ?rt <- (right-turn $?lineList-2 for ?phase)
  =>
  (retract ?lt ?rt)
  (bind ?lenl (length $?lineList-1))
  (bind ?lenr (length $?lineList-2))
(bind ?i 0)
(if (> ?lenl 0) then   ; any left turns
  (while (< ?i ?lenl) do
    (bind ?lno (nth (+ ?i 1) $?lineList-1))
    (assert (temp ?lno for ?phase))
    (bind ?i (+ ?i 1)))
 else (if (> ?lenr 0) then
```

```
    ; no left turns, some right turns
    (while (< ?i ?lenr) do
      (bind ?lno (nth (+ ?i 1) $?lineList-2))
      (assert (temp ?lno for ?phase))
      (bind ?i (+ ?i 1)))))
(if (or (> ?lenl 0) (> ?lenr 0)) then
  (assert (has-same-angle for ?phase)
          (min-angle 63000 to 0 for ?phase))))
          ; 2*pi < 6.30

(defrule calcAngle-1
  (declare (salience -40))
  ?tt <- (temp ?lno for ?phase)
  (t-poly $?pts ?x1 ?y1 ?x2 ?y2 for ?phase)
  (line ?lno ?x2 ?y2 ?x3 ?y3 $?pts1)
  =>
  (retract ?tt)
  (bind ?ax (- ?x1 ?x2)) ; x component of vector a
  (bind ?ay (- ?y1 ?y2)) ; y component of vector a
  (bind ?bx (- ?x3 ?x2)) ; x component of vector b
  (bind ?by (- ?y3 ?y2)) ; y component of vector b
  (bind ?a-len (sqrt (+ (* ?ax ?ax) (* ?ay ?ay))))
  (bind ?b-len (sqrt (+ (* ?bx ?bx) (* ?by ?by))))
  (if (or (= ?a-len 0) (= ?b-len 0)) then
    (bind ?angle 62831)    ; 2*pi = 6.28318
  else
```

```
    (bind ?a-mul-b (+ (* ?ax ?bx) (* ?ay ?by)))
    (bind ?angle (acos (/ ?a-mul-b (* ?a-len ?b-len))))
     ; angle in radians
    (bind ?angle (trunc (* ?angle 10000))))
  (assert (angle ?angle to ?lno for ?phase)))

(defrule calcAngle-2
  (declare (salience -40))
  ?tt <- (temp ?lno for ?phase)
  (t-poly $?pts ?x1 ?y1 ?x2 ?y2 for ?phase)
  (line ?lno $?pts1 ?x3 ?y3 ?x2 ?y2)
  =>
  (retract ?tt)
  (bind ?ax (- ?x1 ?x2)) ; x component of vector a
  (bind ?ay (- ?y1 ?y2)) ; y component of vector a
  (bind ?bx (- ?x3 ?x2)) ; x component of vector b
  (bind ?by (- ?y3 ?y2)) ; y component of vector b
  (bind ?a-len (sqrt (+ (* ?ax ?ax) (* ?ay ?ay))))
  (bind ?b-len (sqrt (+ (* ?bx ?bx) (* ?by ?by))))
  (if (or (= ?a-len 0) (= ?b-len 0)) then
    (bind ?angle 62831)    ; 2*pi = 6.28318
   else
    (bind ?a-mul-b (+ (* ?ax ?bx) (* ?ay ?by)))
    (bind ?angle (acos (/ ?a-mul-b (* ?a-len ?b-len))))
     ; angle in radians
    (bind ?angle (trunc (* ?angle 10000))))
```

```
  (assert (angle ?angle to ?lno for ?phase)))

(defrule findMinAngle
  ?ang <- (angle ?angle1 to ?lno1 for ?phase)
  ?min <- (min-angle ?angle2 to ?lno2 for ?phase)
  ?hsa <- (has-same-angle $?lineList for ?phase)
  =>
  (retract ?ang)
  (if (= ?angle1 ?angle2) then
    (retract ?hsa)
    (bind $?list (mv-append $?lineList ?lno1))
    (assert (has-same-angle $?list for ?phase))
   else (if (< ?angle1 ?angle2) then
    (retract ?min ?hsa)
    (assert (has-same-angle ?lno1 for ?phase)
            (min-angle ?angle1 to ?lno1 for ?phase)))))

(defrule moreTestsNeeded
  (declare (salience -50))
?min <- (min-angle ?angle to ?lno for ?phase)
?hsa <- (has-same-angle $?lineList for ?phase)
=>
(if (= (length $?lineList) 1) then
  (retract ?hsa)
 else
  (retract ?min)
```

```
    (assert (process for ?phase)
            (point-no 3 for ?phase))))

(defrule modifyProcess
  ?pro <- (process $?pt-list for ?phase)
  (point-no ?number for ?phase)
  (has-same-angle $?lineList for ?phase)
  (test (> (length $?lineList) (/ (length $?pt-list) 2)))
  (t-poly $?pts ?tpx ?tpy for ?phase)
  (line ?lno $?lpts)
  (test
     (= (nth (+ (/ (length $?pt-list) 2) 1) $?lineList) ?lno))
=>
  (retract ?pro)
  (bind ?len (length $?lpts))
  (bind ?fx (nth 1 $?lpts))
  (bind ?fy (nth 2 $?lpts))
  (if (and (= ?fx ?tpx) (= ?fy ?tpy)) then
    (if (> (* ?number 2) ?len) then
      (bind ?index (- ?len 2))
    else
      (bind ?index (* (- ?number 1) 2)))
  else
    (if (> (* ?number 2) ?len) then
      (bind ?index 0)
    else
```

```
        (bind ?index (- ?len (* ?number 2)))))
    (bind ?x (nth (+ ?index 1) $?lpts))
    (bind ?y (nth (+ ?index 2) $?lpts))
    (bind $?list (mv-append $?pt-list ?x ?y))
    (assert (process $?list for ?phase)))

(defrule findMostConvex
    ?pro <- (process $?pt-list for ?phase)
    ?num <- (point-no ?number for ?phase)
    ?hsa <- (has-same-angle $?lineList for ?phase)
    (test (= (/ (length $?pt-list) 2) (length $?lineList)))
    (t-poly $?pts ?x1 ?y1 ?x2 ?y2 for ?phase)
    =>
    (retract ?pro ?hsa ?num)
    (bind ?count 1)
    (bind ?min-angle 62831)    ; 2*pi = 6.28318
    (bind ?len (length $?pt-list))
    (bind ?ax (- ?x1 ?x2))
    (bind ?ay (- ?y1 ?y2))
    (bind ?a-len (sqrt (+ (* ?ax ?ax) (* ?ay ?ay))))
    (bind ?i 0)
    (while (< ?i ?len) do
      (bind ?bx (- (nth (+ ?i 1) $?pt-list) ?x2))
      (bind ?by (- (nth (+ ?i 2) $?pt-list) ?y2))
      (bind ?b-len (sqrt (+ (* ?bx ?bx) (* ?by ?by))))
      (bind ?a-mul-b (+ (* ?ax ?bx) (* ?ay ?by)))
```

```
(bind ?angle (acos (/ ?a-mul-b (* ?a-len ?b-len))))
(bind ?angle (trunc (* ?angle 10000)))
(if (= ?angle ?min-angle) then
  (bind ?count (+ ?count 1))
 else (if (< ?angle ?min-angle) then
  (bind ?count 1)
  (bind ?lno (nth (/ (+ ?i 2) 2) $?lineList))
  (bind ?min-angle ?angle)))
   (bind ?i (+ ?i 2)))
  (if (or (= ?count 1) (= ?number 10)) then
   (assert (append-line ?lno for ?phase))
  else
   (assert (process for ?phase)
           (point-no =(+ ?number 1) for ?phase))))

(defrule createAL
  (declare (salience -50))
  ?ma <- (min-angle ?angle to ?lno for ?phase)
  =>
   (retract ?ma)
   (assert (append-line ?lno for ?phase)))

(defrule appendLine-1
  ?al <- (append-line ?lno for ?phase)
  ?tp <- (t-poly $?pts ?x ?y for ?phase)
  (line ?lno ?x ?y $?pts1)
```

```
?bo <- (borders $?lines for ?phase)
=>
(retract ?al ?tp ?bo)
(bind $?stuff1 (mv-append $?pts ?x ?y $?pts1))
(bind $?stuff2 (mv-append $?lines ?lno))
(assert (t-poly $?stuff1 for ?phase)
        (left-turn for ?phase)
        (right-turn for ?phase)
        (borders $?stuff2 for ?phase)))

(defrule appendLine-2
  ?al <- (append-line ?lno for ?phase)
  ?tp <- (t-poly $?pts ?x ?y for ?phase)
  (line ?lno $?pts1 ?x ?y)
  ?bo <- (borders $?lines for ?phase)
  =>
  (retract ?al ?tp ?bo)
  (bind ?len (length $?pts1))
  (bind $?stuff1 (mv-append $?pts ?x ?y))
  (while (> ?len 0) do
    (bind ?xa (nth (- ?len 1) $?pts1))
    (bind ?ya (nth ?len       $?pts1))
    (bind $?stuff1 (mv-append $?stuff1 ?xa ?ya))
    (bind ?len (- ?len 2)))
(bind $?stuff2 (mv-append $?lines ?lno))
(assert (t-poly $?stuff1 for ?phase)
```

```
            (left-turn for ?phase)
            (right-turn for ?phase)
            (borders $?stuff2 for ?phase)))

(defrule checkPoly
  ?tp <- (t-poly ?fx ?fy $?pts ?fx ?fy for ?phase)
  (borders $?borders for ?phase)
  =>
  (retract ?tp)
  (assert (polygon ?fx ?fy $?pts ?fx ?fy for ?phase))
  (printout t "borders of " ?phase " : " $?borders crlf))

(defrule findEutectic
  (polygon $?pts for liquid)
  (lowest-y ?ymin)
  (highest-y ?ymax)
  (min-temperature ?minTemp)
  (advance-in-temp ?advance)
  =>
(bind ?len (length $?pts))
(bind ?minY (nth 2 $?pts))
(bind ?i 0)
(while (< ?i ?len) do
  (if (< (nth (+ ?i 2) $?pts) ?minY) then
    (bind ?minY (nth (+ ?i 2) $?pts)))
  (bind ?i (+ ?i 2)))
```

```
(bind ?m-temp
      (+ (* (* (/ (- ?minY ?ymin) (- ?ymax ?ymin))
               ?advance)
            10)
         ?minTemp))
(assert (min-melting-temp ?m-temp))
(printout t "minimum melting temperature : " ?m-temp crlf))
```

9. Vita

Ali Yıldırım was born on February 5, 1967 in Dursunbey, Turkey. His parents, Abdullah Yıldırım and Zahide Yıldırım, raised him in towns of Çanakkale, a province in Western Turkey. In 1983, he graduated from Çanakkale High School. From Fall 1983 to Spring 1987 he attended Hacettepe University, in Ankara, Turkey, and graduated with a B.S. in Electrical and Electronics Engineering in June 1987. He got the first place among the students who attended the Electrical and Electronics Engineering Department and the second place among the students attending the College of Engineering in the academic year 1986-1987. After a two-month vacation, he started working for the TELETAŞ Telekoműnikasyon Endűstri Ticaret A.Ş. Research and Development Laboratories, one of the two biggest telecommunication companies in Turkey. While working for TELETAŞ, he was also attending Technical University of Istanbul to get his M.S. in Computer Science. Approximately one and a half years later, he won a scholarship from Turkish Ministry of Education and left his country on March 17, 1989 for a M.S. and Ph.D. in Computer Science in the United States of America.