Computer Interpretation of Process and Instrumentation Drawings

Progress Report 1

by

C. Howie, T. Binford, T. Chen, J. Kunz, K. Law

CIFE Working Paper
Number 37

February, 1995

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 94305-4020

Computer Interpretation of Process and Instrumentation Drawings

Progress Report 1

C.Howie, T.Binford, T.Chen, J.Kunz, K.Law

January 1, 1995

A CIFE seed project to build a computer system capable of interpreting Process and Instrumentation Drawings (P&IDs) was initiated in early October of 1994. A prototype methodology for computer interpretation of P&IDs has been developed and implemented on a UNIX *platform. The system is presently able to interpret a portion of an actual P&ID stored in the* DXF *format as well as various small test drawings.*

1 Introduction

This paper introduces the background to the identified problem as well as the project objectives. Thereafter, two case examples demonstrate the capabilities of the current prototype system we have developed. The key concepts of the interpretation methodology are also described.

1.1 Problem Background

Process and Instrumentation Diagrams (P&ID) describe the logical connectivity of components in a process plant. Figure 1 shows a portion of such a drawing. Most lines on the drawing are used to represent pipes or are the outlines of symbols used to depict plant components connected by the pipes. Other lines may be related to text annotations, or used to visually delineate regions of a drawing.

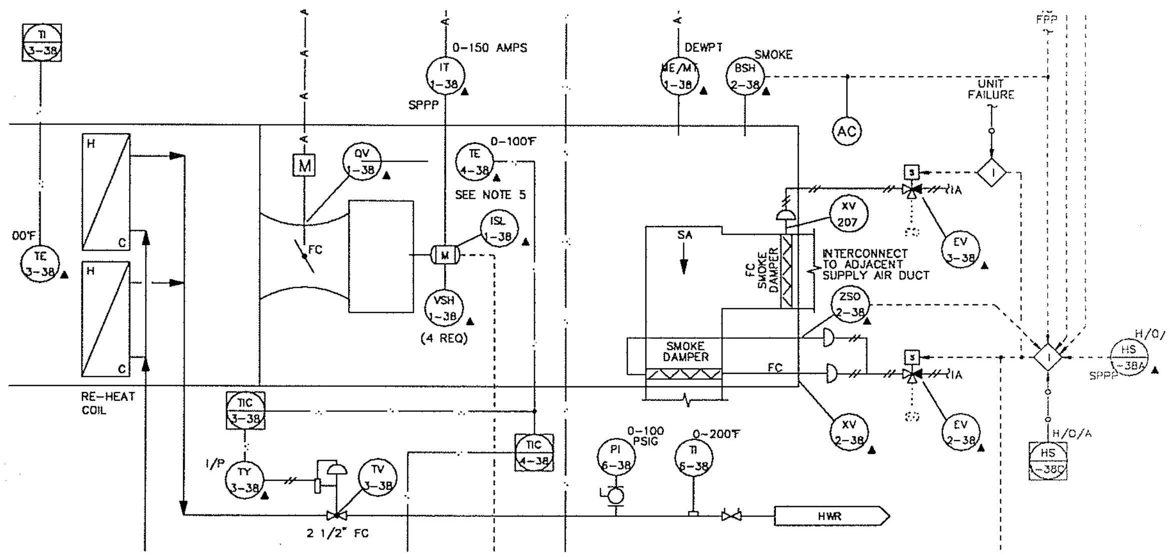

Figure 1: Portion of a Process and Instrumentation Diagram (P&ID). Most lines on the drawing are used to define symbolically the outlines of plant components or to represent pipes connecting the components; some lines are part of text annotations and others may be used to visually divide the drawing into logical sections.

Process plant management, maintenance, and upgrading require frequent reference to the many P&IDs necessary for a typical plant. Modern computerized systems can greatly assist in the analysis and processing of plant component information. Many P&IDs are out of date and have incorrect notations on them as a result of progressive plant maintenance over time. Therefore computerizing the information on the drawing will enable plant operators to ascertain the correctness of component information as well as assisting them in running a plant successfully.

Plant owners now routinely pay upwards of $1000 to have the information of a single P&ID redrawn in an electronic format consistent with their computing resources; most P&IDs are available on either paper or mylar. Even when converted to vector form, such electronic P&IDs can only be edited and analyzed manually, since the drawing system cannot interpret the vectors as functioning components such as pumps, valves, and pipes. Considering that some plants may have several hundred P&IDs, the cost of manually transferring the drawing information to a computerized model can be substantial. A CIFE seed project began in October 1994 in an attempt to build a computer system capable of interpreting P&ID information and so provide plant operators with an automatic facility for the transfer of information from a paper or mylar medium to a symbolic one.

1.2 Long-term Project Objectives

The project research team aims to build a computer system capable of interpreting component connectivity represented graphically in P&IDs. As input, the system will accept either an electronic drawing in the popular DXF format or one on paper or mylar which has been scanned and its raster image converted to vector format. System output will be a symbolic model describing the interpreted plant component connectivity. This model may be a simple one for editing purposes, or one for input to other computer applications for analysis and possibly simulation of plant processes. Whether the input is a DXF or vector-format drawing, the system will be able to identify plant component symbols, their functionality, and their interconnectivity.

The programming language used for system development has not been decided on as yet, though work to date has been done using the C programming language which enables upgrading the system to C++ in the future. It is unlikely that a specific hardware platform will be required for drawing interpretation, and it is expected that a full implementation of the proposed system will be available on common UNIX computers. The final system may well be portable to popular PC platforms.

1.3 Short-term Project Objectives

In the initial phase of the project, we are developing a prototype system which interprets basic P&IDs input in DXF format. Some of the P&IDs are hand-generated in order to test specific interpretation features which a more generic system must support. Section 2 presents two examples of such drawings which demonstrate current system interpretation capability.

We are using an *ad hoc* bottom-up approach to develop a feel for the problem and to provide a platform for scaling the system upwards in order to deal with more complicated drawings. We are currently developing a capability to convert vector information from scanned P&IDs into simple DXF to meet the long-term project objectives with respect to system input.

In addition to the interpretation system, we are also developing auxilliary programs for later integration into the system, and programs for the user to provide information to the system to aid it in interpreting plant component functionality and connectivity details. An interactive feedback feature is being developed to enable us to assess the accuracy of interpretations using the `AutoCAD` system.

2 Case Example

This section presents two drawings to demonstrate the interpretation capability of the current prototype system. This system has been developed using an ad hoc approach, the methodology of which is presented in Section 3.

Figure 2 shows a drawing developed for testing purposes, while Figure 3 shows a section of an Intel Corporation P&ID. Each figure is followed by a table listing the processing time required on a low-end Intel 486 PC computer. With prototyping, code optimization possibilities have been ignored. Many of the algorithms used are highly inefficient; however, real-time interpretation is not within the scope of this research project.

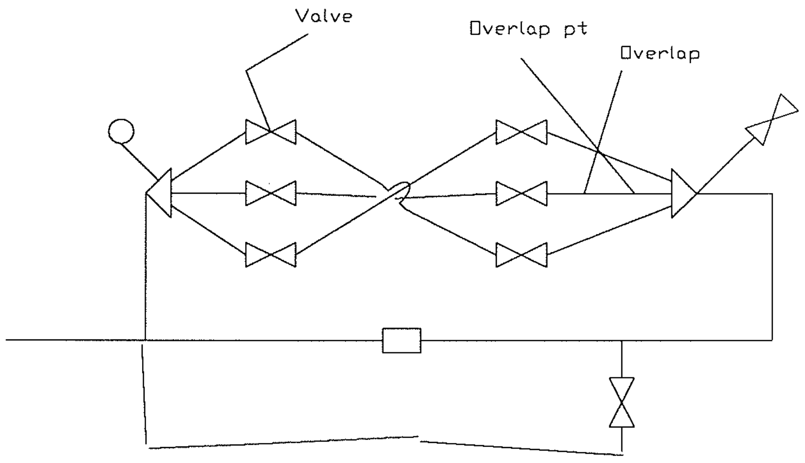

Figure 2: Drawing created to test various features of our interpretation algorithm. The drawing complexity includes lines which overlap (but appear to be a single line), text annotations, different symbols with different degrees of connectivity, lines which appear to have touching endpoints, lines which cross over other lines, and lines which meet at junctions. We have developed a prototype system capable of interpreting the connectivity in this drawing.

Routine	Time (secs)
Parse DXF File	0.22
Overlapping Lines	0.04
Filter Out Text/Irrelevant Lines	0.29
Create Joint Entities	0.16
Merge Lines With Touching Endpoints	16.06
Crossing Lines & Lines Close To Joints	0.19
Connect Blocks	0.12
Connect Joints	0.04
Write Output Files	0.28
TOTAL TIME	17.44

Table 1: Processing times for program running on Intel 486 PC to interpret the above drawing.

3 Interpretation Methodology

This section describes the present methodology designed to interpret initial drawings used for testing purposes. An *ad hoc* approach has been adopted to build a methodology around these simple drawings, an example of which is given in Figure 4.

Test drawings are created with scalability of the resulting method in mind. DXF drawings of actual P&IDs contain too much information for prototype work and would take too long to process repeatedly during system testing. By developing a methodology on small, simple drawings, it will be possible to scale

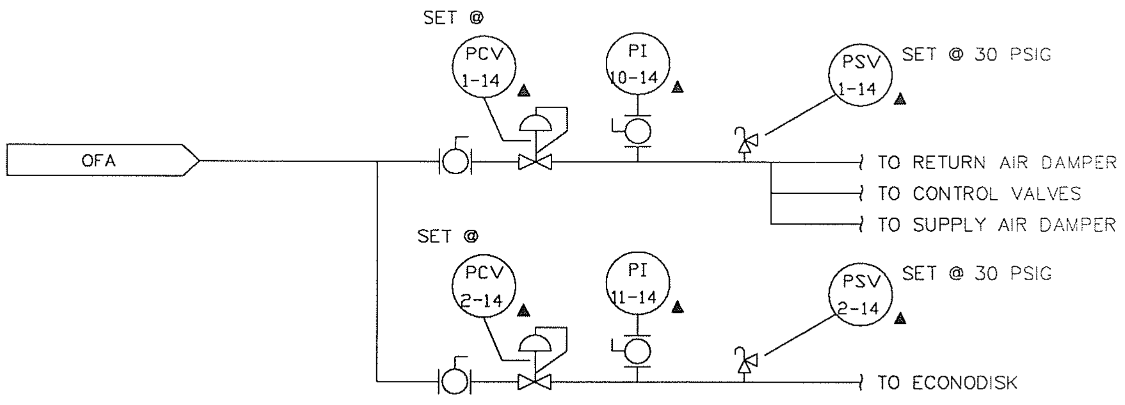

Figure 3: Portion of an actual P&ID, courtesy of Intel Corporation. The current system is able to interpret this portion. Future work will expand the system to enable interpretation of the complete drawing.

Routine	Time (secs)
Parse DXF File	1.07
Overlapping Lines	0.05
Filter Out Text/Irrelevant Lines	0.38
Create Joint Entities	0.09
Merge Lines With Touching Endpoints	21.32
Crossing Lines & Lines Close To Joints	0.15
Connect Blocks	0.08
Connect Joints	0.04
Write Output Files	0.29
TOTAL TIME	23.60

Table 2: Processing times for program running on Intel 486 PC to interpret the above drawing.

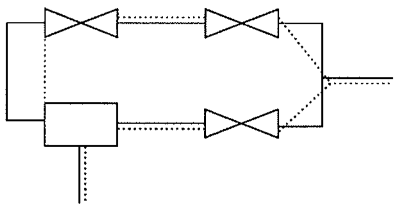

Figure 4: Example of a simple P&ID. Solid lines are either pipes connecting plant components, or outlines of component symbols. The ⋈ symbol is commonly used to represent valves; the rectangle is an arbitrary process component, say a pump. The dotted lines show component connectivity inferred by the P&ID interpretation system.

up towards interpretation of larger ones, with fewer program alterations required at each graduation. Actual P&IDs have been studied in order to construct smaller drawings suitable for this bottom-up approach.
The methodology is diagrammatically shown in Figure 5.

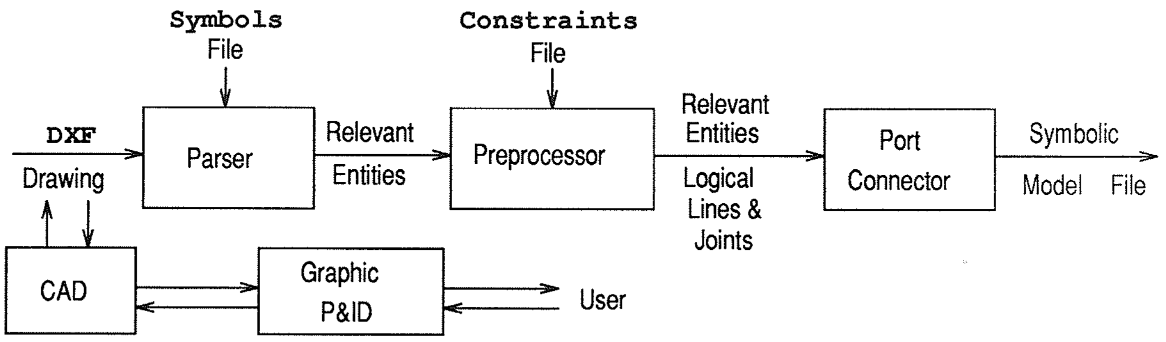

Figure 5: Pictorial summary of present interpretation methodology. Figure 4 shows a graphical representation of a simple input P&ID and Figure 6 shows how that drawing's connectivity would be interpreted.

The input drawing, in DXF format, is parsed and the system builds an internal representation of the drawing *entities*. We define an entity to be either a line in the drawing, a symbolic grouping of lines used to iconify a plant component (called a *block* in DXF syntax), or a string of text. To this list, we add a dynamic entity we call a *joint*. Joints are created by the system interpreter at the intersection of pipes to facilitate the compilation of a symbolic model supporting pipe splitting.

The user provides a *symbol* file to enable the system to filter out any symbolic entities irrelevant to the interpretation process. A preprocessing stage, with the aid of user specifications provided in a *constraint* file, simplifies drawing details where possible. Thereafter, the connectivity between relevant drawing entities is inferred, and a symbolic model output. Figure 6 shows how the simple P&ID in Figure 4 would be interpreted.

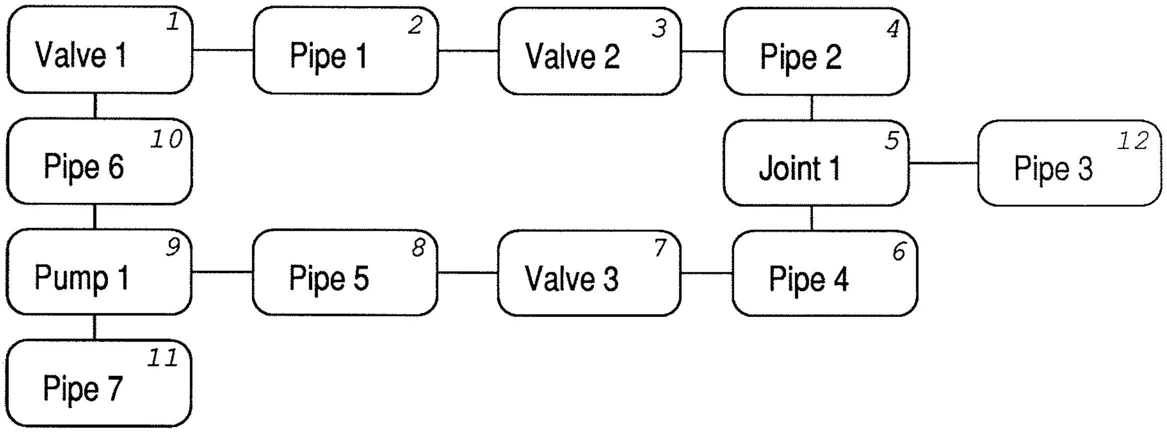

Figure 6: Symbolic model output by the P&ID interpretation system for the simple drawing in Figure 4. The numbers in the top-right corner of each box are entity reference numbers.

The methodology stages are discussed in more detail below.

3.1 Input Drawings

We chose the DXF format for initial system testing and development for three reasons. It is the most widely supported format for electronic drawings, drawings for testing purposes can readily be created in this format using available CAD systems, and DXF drawings can be downgraded to raw vector information which enables us to test the system on vector format drawings. A vector drawing does not include the symbolic grouping of line primitives into *blocks* as supported by the DXF format.

The current computer program supports only DXF drawings as these inherently provide information useful to assist in interpretation of drawing entities and their interconnectivity. Support for vector format drawings will be added in the future. If it is possible to convert a vector format drawing to a DXF equivalent, then building the system around the DXF format will meet the project objectives with respect to system input as vector drawings could be converted to DXF before interpretation processing.

The research team has several P&IDs from Intel Corporation in DXF format. It is in the process of trying to acquire a scanned P&ID which has been converted to simple vector format. However, since a vector format is a subset of DXF, it is possible to downgrade a DXF drawing to test a methodology on vector input.

3.2 Additional Input Files

Before the program attempts to determine the connectivity of the entities in the drawing, user specifications allow both the filtering out of entities of no relevance to the interpretation process and the conversion of groups of line entities into equivalent *logical* lines. Figure 4 demonstrates how the original lines in a drawing (shown as solid lines) can be converted to logically equivalent lines (shown dotted) where possible. We use logical lines to simplify the inference of component interconnectivity.

3.2.1 The `symbols` file

Many symbols on P&IDs convey information superficial to the component connectivity described. The symbolic model which the system attempts to build requires that only certain symbols in the drawing have their connections understood. Text annotations are irrelevant, as might be other drawing symbols. In order to build a successful model it is, therefore, necessary to determine only which line entities represent pipes and which drawing symbols represent plant components connected by these pipes.

With a symbol file, the user can define which graphical symbols in the DXF input drawing are relevant for inclusion in the output symbolic model. The system then ignores any irrelevant symbols and any symbols which are not listed in the file. The latter requirement arises from the fact that DXF files output from CAD systems will usually contain meta-data *blocks*, each with its own identification label, which are of no use to the interpreter. DXF drawings are frequently drafted with block identifiers. A block groups several drawing primitives (lines, arcs, circles) into a single entity for reference purposes, and so defines a graphical object. A drafter can then insert instances of the generic symbol in the drawing, at any specified scale and rotation angle. We take the term 'block' from the terminology used in DXF drawing syntax. Figure 7 shows how the four line primitives for a *valve* symbol might be grouped into a block labelled VLV.

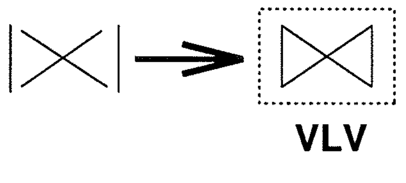

Figure 7: An identifier can be used to refer to a group of *blocked* drawing primitives (lines, arcs, circles).

The user provides a computer file, called the **`symbols`** file, that lists all the block labels which have been used to identify drawing entities. The symbols file is also used to specify basic information about block symbols, including whether or not a block entity should be considered for connectivity interpretation. This information is then used for inferencing purposes when entity connections are established. Any lines which are part of, or touch, specified irrelevant symbols are ignored by the program for connectivity purposes. The parsing routine which reads drawing data in from the DXF file bypasses meta-data, filtering out only the drawing entity blocks, including blocks which the user specifies as being irrelevant.

3.2.2 The constraint file

The user-specified `constraint` file defines the distance tolerances used for inferencing as described below. Such tolerances are used to resolve ambiguity in situations in which lines touch one another or a block symbol outline, or in which line endpoints are in close proximity to other lines and their endpoints. Figure 8 shows examples where tolerances are used in this manner.

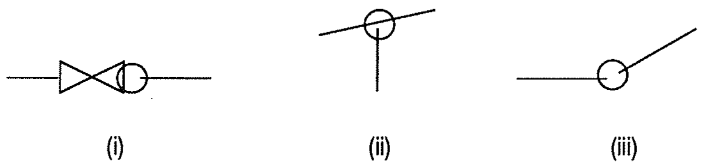

Figure 8: Three examples in which it is not certain whether a line should be interpreted as touching another line close by. User-specified distance tolerances are used to resolve such ambiguity. Multiple tolerance parameters are used to support ambiguity resolution in different situations. For example, case (i) shows a line entity near a process component symbol, while cases (ii) and (iii) show lines near other lines; the user may prefer to use a different tolerance value for each case.

3.3 Drawing Entity Preprocessing

3.3.1 Logical lines

Line entities in a P&ID represent either the outlines of symbols depicting plant components or pipes which connect these components. For drawing interpretation purposes, it is more efficient to convert several physical lines representing a pipe into a single *logical* line for that pipe. This is demonstrated in Figure 9 where the three pipe segments connecting two valve entities can be replaced with a single equivalent logical pipe, shown as a dotted line. Preprocessing a drawing includes the creation of such logical lines where appropriate.

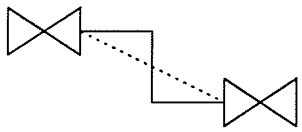

Figure 9: Physical line entities (shown as solid lines) are converted to an equivalent *logical* line entity (shown as a dotted line). To preserve symbolic information, the solid lines of the valve (⋈) symbols cannot be converted.

3.3.2 Imprecise line data

It is not uncommon to find two lines in a DXF file which overlap one another and so appear as a single line to the human eye on the actual drawing. For example, the two lines in a Cartesian coordinate system defined by endpoints (3,5)-(10,5) and (6,5)-(12,5) will appear as a single line (3,5)-(12,5). However, in processing the entities in the DXF file, it is convenient that such overlapping lines be replaced with a single line defined by the two extreme points. Preprocessing of drawing entities makes such replacements.

A sequence of line segments might also have endpoints which do not quite touch, but which, for connectivity purposes, we consider as touching. Figure 10 shows end xample case in which lines with endpoints almost touching are replaced by a single equivalent logical line defined by the two extreme endpoints.

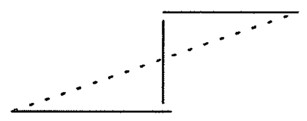

Figure 10: Several lines with endpoints in close proximity are replaced by a logical (dotted) line defined by the two extreme endpoints.

3.3.3 Joint entities

Where pipes come together to form an intersection, it is necessary to create a *joint* entity in order to assemble a symbolic model of all entity connections. Such joint entities do not exist in the original DXF file and so are created during the preprocessing phase. Figure 11 shows how a four-way joint would be created at the intersection of three pipe segments. In this example, one of the pipe segments does not quite touch the intersection point, but the system will infer that it does by using a distance tolerance parameter specified by the user.

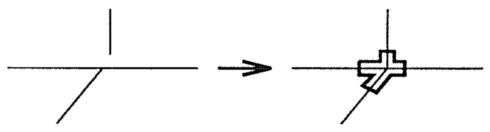

Figure 11: A four-way joint is created at the intersection of three pipe entities. Imprecise placement of line endpoints can be resolved using specified distance tolerance parameters.

3.3.4 Crossing lines

In Figure 12, line endpoints within a specified distance tolerance are used to interpret *crossing* lines. Such crossing lines are replaced by a single logical line, shown as a dotted line in the figure. The interpretation algorithm uses the Euclidean distance between endpoints to support crossing lines which are not colinear.

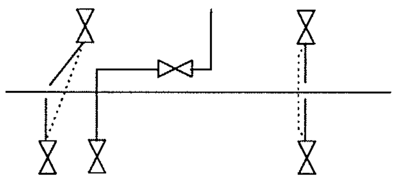

Figure 12: Interpretation of *crossing* lines where line endpoints are sufficiently close, and their other endpoints connected to relevant symbolic drawing entities.

3.4 Text annotation lines

A line with an endpoint within a specified tolerance of a text entity is considered to be an annotation line associated with the text, and is ignored when determining entity connections. This is demonstrated in Figure 13.

At present, the location of the text entity is considered by the program to be simply the Cartesian coordinate at which the text entity was inserted in the drawing. Future improvement of the program will

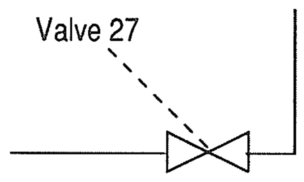

Figure 13: A line entity (shown as a dashed line) is considered to be associated with a text annotation if one of its endpoints lies within a specified distance of a text entity. This separates the line from the set of line entities which will be inferred as pipes.

require a *bounding box* of the text dimensions to be considered for the purpose of dismissing annotation lines.

3.5 Determination of Entity Connectivity

Once the above preprocessing of drawing entities is complete, the program establishes the connectivity between the relevant entities.

3.5.1 Entity ports

The present methodology considers each entity to have logical *ports* to which line entities depicting pipes are attached. For example, a *valve* entity will always have two pipes connected to it. A valve will therefore have two ports, each providing an attachment point for one endpoint of a pipe entity. The `symbols` file allows the user to specify the number of ports for the different entity types in the drawing. All pipe entities are automatically assigned two ports, one at each end of the pipe.

At present, the program makes no attempt to constrain the geometric location of entity ports with respect to the symbol icon. For example, Figure 14 shows the two dotted line primitives of a valve entity on which, by convention, one would expect to find its two logical ports.

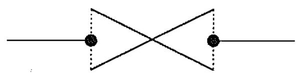

Figure 14: Port attachment points ideally should be constrained to specific primitives used to define a drawing symbol. The present interpretation algorithm does not constrain any entity ports to specific symbol primitives.

The program presently considers the location of ports for non-pipe entities to be the Cartesian point at which the entity is located in the drawing. Thus the system will consider a symbolic entity assigned more than one port to have all these ports at the same position on the symbol. To address this issue and so provide a more knowledge-based approach in determining entity connectivity, a database program has been developed to allow the user to specify, for each entity in a separate database, the particular entity primitives on which that entity's ports are located. This database will improve the interpretation capability of the existing program when processing more complicated P&IDs, though it has not been required on test drawings used to date.

Joint entities have as many ports as there are pipes connected to them. The location of all these ports is the point of intersection of the applicable pipes.

3.5.2 Entity proximity lists

Once the Cartesian coordinates of all entity ports have been determined, a proximity list (of fixed length) of the nearest ports can be built for an entity port. We currently restrict the lists to hold information for 8 neighboring ports. While this arbitrary number may seem large, in connecting pipes to entity ports, some pipes may be rejected for logical reasons (for example, the two endpoints of a pipe are unlikely to be connected to the same component entity) and so the proximity lists must be long enough to provide sufficiently many candidate ports for attachment. A proximity list is sorted on the Euclidean distance to neighboring ports of other entities, so that the closest ports are selected as the most likely pipes for connection to the given entity.

3.5.3 Entity connections

An iterative routine loops through all the entity ports which are currently unconnected. For each port, the system builds a proximity list as described above. A port is then connected to the nearest port in its proximity list. If, for logical reasons, it is undesirable to connect to the first port in a proximity list, the next most-suitable port in the list is chosen. For each entity port, the system records the reference number of the pipe entity, and that pipe's port, which is attached to the port. For example, a valve with the system reference number 22 might have its first port connected to the second port of a pipe with reference number 47; the system then records entity 22's port 1 as being connected to entity 47's port 2. This syntax implicitly defines the symbolic model which will be output by the system.

For consistency, and to reduce computation time, the port on the second entity can be connected while processing the port on the first entity. For the above example, this means entity 47's port 2 is recorded as being connected to entity 22's port 1 at the same time as the port on entity 22 is being resolved. This obviates the need to construct a proximity list and look for a connection when the iteration loop reaches the second port of entity 47.

3.6 Program Output

The program outputs a list of all relevant entities and their ports, indicating for each one the reference numbers of the entity-port connection with another entity. This format implicitly defines the connection model of all relevant drawing entities. The model is written to a file so that it can be input to other computer applications. The current system can also generate an `AutoLisp` feedback file which provides interactive display of the interpreted connectivity to the user through the `AutoCAD` system.

4 Implementation

This section describes the current implementation of the methodology.

4.1 Program Overview

The methodology has been coded using the C programming language running on a UNIX system. Specifically, the program has been installed on a CIFE Sun workstation and an Intel 486 PC running a UNIX operating system. While some features are specific to a UNIX platform, these could be ported to other operating system environments. The program currently only supports input drawings which are in the DXF format, and is run from the system command line. Various optional parameters are supported to control the degree of feedback to the user, both for on-screen display and the types of output files.

4.1.1 Constraint file

The user can specify the line entity proximity constraints using a constraint file which is in ASCII format. This allows the `constraint` file to be prepared using any text editor. 'Comment' lines can be inserted in the file for readability. An example of a constraint file is given below, with comment lines using a leading '#' character. Each tolerance parameter is explained briefly with comment lines appearing before it.

```
# Global constraint variables
#-------------------------------------
# Greatest strline dist between end of a line and part of another line
# when doing connection proximity calculations. This is the VERY CLOSE
# proximity tolerance for joints and lines touching blocks.
# This distance is also taken as the tolerance for 'tap' blocks which
# tap into pipes.
LINE_PROXIMITY_ERROR,0.05
#---------------------------------------
# Greatest strline dist between line endpoints tolerated for line merging
# Also used to make joints
LINE_ENDPOINT_ERROR,0.1
#---------------------------------------
# This is tolerance for crossing lines
LINE_JUMP_ERROR,0.1
#---------------------------------------
# Max number of connetion ports any entity in drawing can have
MAX_PORTS_PER_ENTITY,10
#---------------------------------------
# Max length of the proximity list which will be built for each entity
MAX_PROXIMITY_NODES,8
#---------------------------------------
# Tolerance for text entities being near line endpoints
TEXT_PROXIMITY_ERROR,0.2
```

4.1.2 Symbol table

The `symbols` file is also implemented using an ASCII format. For each symbol which appears in the DXF drawing, the user specifies how pipes will attach to the symbol, a basic description for the symbol (this simply improves feedback and understanding of the computer interpretation – it plays no role in any inferencing routines), and the identifier label used to define the symbol as a DXF *block* entity (where appropriate). The user also specifies whether or not the symbol is relevant to the connectivity model which will be assembled, and the maximum number of attachment ports (see Section 3.5.1) that the symbol has available for connecting pipe entities to. To demonstrate the implementation of the `symbols` file, below is the file used for interpretation of the P&ID in Figure 3.

```
# type,label,description,relevant(Y/N)?,max num ports
LINE,NULL,NULL,NULL,Y,2
BLOCK,INTERNAL,80VL01,Valve,Y,2
BLOCK,EXTERNAL,80LI07,Air source,Y,1
BLOCK,EXTERNAL,80VL41,Valve cap,N,1
BLOCK,INTERNAL,V_BALL,Funny round looking object,Y,2
BLOCK,TAP,V_PRELEF,Also funny looking thing,Y,1
BLOCK,EXTERNAL,60IN,Text annotation symbol,N,1
BLOCK,EXTERNAL,END,Text hook,Y,1
```

4.2 Input File Parser

The parsing routine extracts all the block definition information for drawing symbols which are identified using DXF block labels. This provides access to the drawing primitives which define each symbol. It then constructs an internal data representation of the lines, symbols and text entities in the drawing. Any symbols whose labels are not found in the `symbols` file are ignored, which prevents the parser from processing meta-data embedded in the DXF file.

4.3 Preprocessing Routines

Once the drawing entities have been parsed, various routines perform the preprocessing of Section 3.3. This reduces the number of line entities. While adding to interpretation overhead (witness the times in Tables 1 and 2), preprocessing greatly simplifies the algorithm for building the connectivity model. Correct sequencing of the preprocessing routines is important otherwise some routines will undo the work of others. Pseudo-code for the preprocessing stage is:

```
{ Look for any two lines which overlap one another }
For i = 1 to number_of_lines
    For j = 1 to number_of_lines
        if i <> j then begin
            if line(i) overlaps line(j) then MakeNewLine(i,j)
            remove line(i)
            remove line(j)
        end

{ Exclude text annotation lines and lines which touch irrelevant symbols }
For i = 1 to number_of_lines
    if line(i) has an endpoint close to text or which touches an
    irrelevant block symbol, then line(i) cannot be a 'pipe' so
    flag it

{ Make 'joint' entities where several pipes intersect }
For i = 1 to number_of_lines
    if line(i) is a pipe and has an endpoint touching another pipe,
    then make a joint at the point of intersection

{ Merge a sequence of lines which have touching endpoints }
For i = 1 to number_of_lines
    For j = 1 to number_of_lines
        if i <> j then begin
            if line(i) has an endpoint which touches an endpoint of
            line(j) then MakeEquivalentLine(i,j)
            remove line(i)
            remove line(j)
        end

{ Resolve crossing lines and lines close to joints }
For i = 1 to number_of_lines
    if line(i) has an endpoint close to an existing joint entity, then
    consider it to be connected to the joint; otherwise

    if line(i) does not touch a relevant block symbol then begin
        For j = 1 to number_of_lines
            if i <> j then begin
                if line(i) has an endpoint close to line(j) then lines
                i and j must be crossing lines so
                MakeCrossingLine(i,j)
                remove line(i)
                remove line(j)
            end
```

4.4 Connectivity Routines

After line and joint preprocessing, the program iterates through the block and joint entities determining, for each entity port, the closest ports of other line entities in close proximity which could be attached to the port. These line entities will be lines representing pipes in the P&ID. Their nearby ports are recorded in *proximity* lists, one for each port.

Once the proximity lists for an entity have been established, each entity port can be connected to the nearest port in its proximity list, unless it is logically unwise to do so. The only case where the current system rejects the first port in the list is if that pipe's other port is already connected to the symbolic entity (block or joint) being processed. If the first port is rejected, the next port in the list is chosen. We presently allow the proximity lists to record the eight nearest ports as future resolution of connectivity ambiguity might well require more complicated rejection of the first few ports in a proximity list.

Each pipe has a port at its ends, but pipe entities need never be connected explicitly as once all the block and joint entities have had their ports processed, there cannot be pipe entities left unattached which are meant to be attached. This is a positive side-effect of line preprocessing and the consistency requirement of Section 3.5.3. The pseudo-code for connecting drawing entities is:

```
{ Determine the 10 nearest ports for a given entity port }
BuildProximityList for entity(i):
For p = 1 to number_of_ports for entity(i)
    if port(p) not already connected then
        For j = 1 to number_of_entities
            For q = 1 to number_of_ports for entity(j)
                if Euclidean distance between entity(i).port(p) and
                entity(j).port(q) is less than the 10 current closest
                ports in the proximity list for entity(i).port(p), then
                insert entity(j).port(q) in this list.

{ Connect the ports of a given entity }
ConnectPorts for entity(i):
For p = 1 to number_of_ports for entity(i)
    if port(p) not already connected then begin
        { Choose a nearby port }
        For j = 1 to number_of_lines
            if line(j) is a pipe and
            if line(j) has no endpoint already attached to entity(i) then
                attach entity(i).port(p) to the nearby port of line(j)

{ Connect block symbol entities }
For i = 1 to number_of_entities
    if entity(i) is a block symbol then begin
        BuildProximityList for entity(i)
        ConnectPorts for entity(i)

{ Connect joint entities }
For i = 1 to number_of_entities
    if entity(i) is a joint then begin
        BuildProximityList for entity(i)
        ConnectPorts for entity(i)
```

4.5 Output Files

The program currently outputs four files in ASCII format. The first is the connectivity file which lists all the drawing entities and their ports. For each port, a corresponding entity and port number are given. This syntax implicitly defines the desired connectivity model. In addition to this connectivity output file, the program also writes processing 'trace' information to a trace file for code development purposes. A thorough output file which lists not only entity port connections, but other entity details, is also output for code development purposes. Finally, a text file using the `AutoLisp` language can be generated for interactive feedback of the interpreted drawing connectivity using the `AutoCAD` system. This feedback goes through each entity in the drawing, one by one, and highlights the pipes to which the entity is considered connected. The visual feedback provides instant conformation of whether or not the methodology correctly interpreted the given drawing without the user laboriously having to inspect the symbolic model in the output file.

To provide an example of a symbolic model output file, below is listed the file generated after interpretation of the P&ID in Figure 3. The data fields are all comma-separated. The first field records the entity reference number. The second field indicates the type of entity, the third and fourth describe each entity. The fifth field is the number of ports for that entity, and thereafter, each port connection has its entity-port numbers listed. The entity-port pair `0,1` indicates that port remained unconnected.

```
6,LINE,null,IRRELEVANT,2,0,1,0,1
8,LINE,null,PIPE,2,40,1,42,1
10,LINE,null,IRRELEVANT,2,0,1,0,1
11,LINE,null,IRRELEVANT,2,0,1,0,1
12,LINE,null,PIPE,2,45,1,68,1
13,LINE,null,IRRELEVANT,2,0,1,0,1
14,LINE,null,PIPE,2,69,1,56,1
15,LINE,null,IRRELEVANT,2,0,1,0,1
18,LINE,null,PIPE,2,50,1,51,1
19,LINE,null,IRRELEVANT,2,0,1,0,1
20,LINE,null,IRRELEVANT,2,0,1,0,1
21,LINE,null,PIPE,2,54,1,70,1
22,LINE,null,IRRELEVANT,2,0,1,0,1
26,LINE,null,PIPE,2,66,1,57,1
27,LINE,null,PIPE,2,37,1,67,1
28,LINE,null,PIPE,2,67,2,50,2
29,LINE,null,PIPE,2,42,2,68,2
30,LINE,null,PIPE,2,68,3,47,1
31,LINE,null,PIPE,2,66,2,69,2
33,LINE,null,PIPE,2,51,2,70,2
34,LINE,null,PIPE,2,70,3,66,3
35,LINE,null,PIPE,2,40,2,67,3
36,LINE,null,PIPE,2,69,3,38,1
37,External,80LI07,Air source,1,27,1
38,External,END,Text hook,1,36,2
40,Internal,V_BALL,Funny thing,2,8,1,35,1
42,Internal,80VL01,Valve,2,8,2,29,1
44,Tap,V_PRELEF,Also funny thing,0,30
45,Internal,V_BALL,Funny thing,2,12,1,0,1
47,External,END,Text hook,1,30,2
50,Internal,V_BALL,Funny thing,2,18,1,28,2
51,Internal,80VL01,Valve,2,18,2,33,1
54,Internal,V_BALL,Funny thing,2,21,1,0,1
55,Tap,V_PRELEF,Also funny thing,0,34
56,External,END,Text hook,1,14,2
57,External,END,Text hook,1,26,2
66,JOINT,null,3-way,3,26,1,31,1,34,2
67,JOINT,null,3-way,3,27,2,28,1,35,2
68,JOINT,null,3-way,3,12,2,29,2,30,1
69,JOINT,null,3-way,3,14,1,31,2,36,1
70,JOINT,null,3-way,3,21,2,33,2,34,1
```

4.6 Database Program

The database program mentioned in Section 3.5.1 has been completed sufficient to allow short-term integration into the existing program. Test drawings used to date have not required its more knowledgeable description of drawing symbol connectivity possibilities, but integration might well be required in the near future as more complicated drawings are processed.

5 Discussion and Future Work

We have made significant progress towards building a computer system capable of interpreting the connectivity in P&IDs. Our decision to use DXF drawings for the initial prototyping enables rapid generation of test drawings using a CAD system. By adopting an ad hoc approach to developing an interpretation methodology, we can create test drawings with specific features and complexity for a particular stage in the project.

The system cannot interpret any of the complete P&IDs we have at present. This is because some of the features in these drawings have not been drawn using *block* information. Consequently, the system is incapable of differentiating symbols comprising raw line entities from lines representing pipes or text annotation lines; it can only recognise a symbol if the primitives defining the symbol have been grouped using a block identifier.

Once we have a working methodology for conversion of common process plant symbols from vector format to simple DXF, we will upgrade the system in order to process a complete P&ID. Concurrently with this work, we are increasing the robustness of the interpretation algorithm and extending it to support additional drawing features. We will also be integrating the database facility described in Section 4.6 with the present system.

6 Conclusion

A program has been written to interpret basic P&IDs which are input in the DXF format. The program implements an ad hoc methodology designed around carefully chosen test drawings. The program is capable of correctly interpreting the connectivity of elements in purpose-built test drawings and a portion of an actual P&ID. The current methodology should scale up to larger drawings without difficulty where these drawings assume a detail complexity consistent with test drawings used to date. We hope to process complete, real P&IDs within a few months, once the methodology has been expanded to support greater drawing complexity and once we can convert vector information to simple DXF. Feedback to the user of program performance and the interpreted symbolic model is provided by way of several output files. The program code has been kept fairly portable as a final implementation computer platform has not been decided upon.