COMPUTER VISION AND IMAGE UNDERSTANDING
Vol. 70, No. 3, June, pp. 420–437, 1998
ARTICLE NO. IV980683

A Generic Integrated Line Detection Algorithm and Its Object–Process Specification

Liu Wenyin* and Dov Dori†

Faculty of Industrial Engineering and Management, Technion—Israel Institute of Technology, Haifa 32000, Israel

Received February 7, 1997; accepted December 15, 1997

A generic integrated line detection algorithm (GILDA) is presented and demonstrated. GILDA is based on the generic graphics recognition approach, which abstracts the graphics recognition as a stepwise recovery of the multiple components of the graphic objects and is specified by the object–process methodology. We define 12 classes of lines which appear in engineering drawings and use them to construct a class inheritance hierarchy. The hierarchy highly abstracts the line features that are relevant to the line detection process. Based on the "Hypothesis and Test" paradigm, lines are detected by a stepwise extension to both ends of a selected first key component. In each extension cycle, one new component which best meets the current line's shape and style constraints is appended to the line. Different line classes are detected by controlling the line attribute values. As we show in the experiments, the algorithm demonstrates high performance on clear synthetic drawings as well as on noisy, complex, real-world drawings. © 1998 Academic Press

Key Words: **line detection; dashed line detection; arc segmentation; object–process methodology; object–process diagrams; graphics recognition; line drawings; engineering drawings interpretation; raster-to-vector conversion; CAD conversion.**

1. INTRODUCTION

Lines are the most basic graphical primitives in line drawings in general and in engineering drawings of all categories in particular. Humans find it easy to recognize the styles and shapes (geometric forms) of lines, because their well-developed visual perception looks simultaneously at related areas and can even "fill the gaps" between consecutive segments. Machines do not possess this natural ability and therefore the task of recognizing lines of various styles (especially discontinuous lines) and shapes by machines is nontrivial, as we show in this paper. In engineering drawings, lines are assigned meanings through different attribute value combinations of their thickness, style, and shape. For example, in mechanical engineering, the line *thickness* attribute is used to differentiate between object contour and annotation graphics. Different drawing standards define different groups of line width values. The ISO drawing standard [1] requires that the line width be within the range of 0.18∼2.0 mm, while the Indian Standard Institute (ISI) standard [2] requires that the line width be from of 0.1∼1.2 mm. However, all drawing standards [1–4] define only two legal line thickness values: *thick* and *thin*, where the width of a thin line is somewhere between 1/3 and 1/2 of the width of a thick line.

The line *style* refers to the continuity of the line. The four possible style values are solid (continuous), dashed, dash–dotted, or dash–dot–dotted. The style alone, or the combination of style and thickness, define the *function* or *semantics* of the line. The function of each combination in the ISO drawing standard [1] is listed in Table 1. Figure 1 demonstrates typical applications of different line semantics. Other drawing standards [2–4] define the line semantics in a similar manner. Usually, thick solid lines represent visible contours in 2D projections of 3D objects, while thin solid lines are mainly dimensioning and leader lines used for annotating these objects. Dashed lines delineate hidden contours of geometrical objects. Dash–dotted lines represent symmetry axes of symmetrical objects. Finally, dash–dot–dotted lines, which are less frequently used in engineering drawings, serve as auxiliary lines. ISI [2] does not even define this style.

The line *shape* defines mainly the geometry of the 2D orthographic projection of the 3D objects. It can be straight, circular, elliptic, free curved, or any other geometric shape. The term *line detection* pertains to the identification of all the three attribute values of the lines, i.e., the thickness, the style, and the shape.

In view of the different functions and semantics of the various line thickness, styles, shapes, and lexical ambiguities (e.g., C1 and D1, E1 and F1, E2 and F2), correct recognition of all these line attributes is an important prerequisite for high-level interpretation of engineering drawings. Line detection has been a heavily researched subject in the context of engineering drawings interpretation during the past 20 years, and many algorithms and systems for line detection have been developed [5–11]. The solid straight line segment, which we call *bar*, is the most common line object in engineering drawings. It is only natural that its recognition has been the primary objective of the vast majority of the line-detection algorithms. The detection of this class of lines is usually done by the vectorization process. Postprocessing refinement is employed by some of these algorithms to improve the bar detection accuracy. Solid polylines, which may be either

* E-mail: liuwy@ie.technion.ac.il.
† E-mail: dori@ie.technion.ac.il.

TABLE 1
Line style Defined and used by ISO Standard Engineering Drawings

Line code and center style	Description	General Applications
A ————————	Continuous thick	A1: Visible outlines A2: Visible edges
B ————————	Continuous thin	B1: Imaginary lines of intersection B2: Dimension lines B3: Projection lines B4: Leader lines B5: Hatching lines B6: Outlines of revolved sections in plane B7: Short center lines
C ～～～～	Continuous thin free form	C1: Limits of partial or interrupted views and sections, if the limit is not a dash-dotted thin line
D —⌇——⌇——⌇—	Continuous thin (straight) with spikes	D1: Similar to C1
E —— —— —— ——	Dashed thick	E1: Hidden outlines E2: Hidden edges
F —— —— —— ——	Dashed thin	F1: Hidden outlines F2: Hidden edges
G ——— - ——— - ———	Dash-dotted thin	G1: Center lines G2: Lines of symmetry G3: Trajectories
H ┗-·-·-┛ ┏-·-·-·—	Dash-dotted thin, thick at ends and changes of direction	H1: Cutting planes
J ——— - ——— - ———	Dashed thick	J1: Indication of lines or surfaces to which a special requirement applies
K ——— -- ——— -- ———	Dash-dot-dotted thin	K1: Outlines of adjacent parts K2: Alternative and extreme positions of movable parts K3: Centroidal lines K4: Initial outlines prior to forming K5: Parts situated in front of the cutting plane

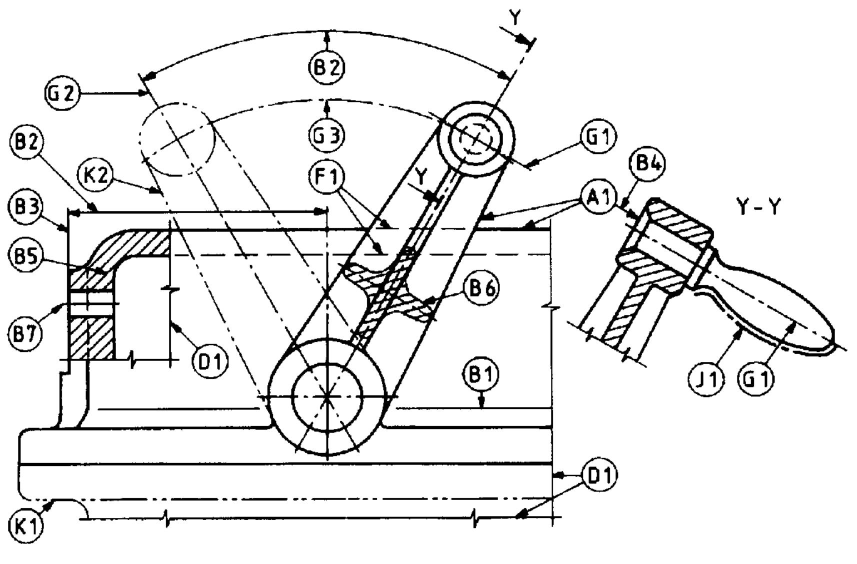

FIG. 1. Illustration of the use of various of line types in ISO standard engineering drawings [1].

real polylines, formed by chaining of bars, or approximations of curved lines, also result from vectorization. Refinement is required if the polyline is an approximation of a curve.

Bars and polylines are relatively easy to detect, while arcs are more difficult, due to their complex geometry. For example, bars can be easily segmented from the run-length coded image in the first two steps (first long horizontal bars and then thin vertical bars) of the four consequential steps for music scores recognition [11]. However, although quite a few arc segmentation algorithms have been developed, e.g., [12–14], the task still seems be to a tough problem. The algorithm in [12] employs Hough transform (HT), which is a conventional method for object extraction from binary images. HT is normally used for arc segmentation in case of isolated points that potentially lie on circles or circular arcs. HT's high complexity in both time and space makes such arc segmentation algorithms less practical for engineering drawings. The algorithm in [13] belongs to the curvature estimation methods. Motivated by object recognition, the aim of these algorithms is to extract meaningful features from objects by estimating their edge curvature. To produce the desired input of the one-pixel-wide digital curve, curvature estimates require heavy, pixel-based preprocessing, such as edge detection or thinning. Algorithms such as [12] and [13] cannot detect the line thickness. Perpendicular bisector tracing (PBT) [14] is the first vector-based method of arc segmentation. As it examines only the bar fragments output by the vectorization process, it is efficient in both time and space. All arc segmentation algorithms discussed above are designed to segment only solid arcs. Research on the detection of arcs of other styles is rare. In the process of arc segmentation reported in [17], Chen *et al.* use some patterns and clues that the line segments, vectorized from an arc, constitute a chain of pseudo-line segments (PLS) that are shorter than some statistical threshold and are delimited between two long straight lines.

Line style detection has also been studied by several groups [7–10, 15–19], but it is treated only as a small, side issue in these works. Pao *et al.* [15] use a HT-based method to detect dashed circles and dashed straight lines in several steps. This pixel-based method segments one class of dashed lines in each step and it is computationally expensive. Boatto *et al.* [8] use a semi-vector-based method to find dash segments which have special graph structures. Other groups use vector-based algorithms to detect discontinuous lines. Vaxiviere and Tombre's Celesstin system can detect both dashed lines and dash–dotted lines according to the French Standard NF E 04-103 [9]. Joseph and Pridmore [10] have dealt with finding dashed lines in engineering drawings by looking for chains of short lines within the ANON system. Lai and Kasturi [16] have done work on detecting dashed lines in drawings and maps. They attempt to recognize dashed lines by linking short isolated bars under certain conditions in three passes. The dashed lines are not necessarily straight, as is the case in maps. Chen *et al.* [17] use the same method as in [16] to detect dashed lines of several patterns in the refinement of vectorized mechanical drawings. Agam *et al.* [18] and Dori *et al.* [19] have recently investigated the detection of dashed and dash–dotted lines with straight and curved shapes. The algorithm in [18] is pixel-based. The image of dashes is first separated from the drawing image and is then processed using a set of tube-directional morphological operators to label the dashed lines. The algorithm in [19] is vector based; it applies the Sparse Pixel Vectorization algorithm [20] as a preprocessing step to produce solid vectors from image drawings. The algorithm examines only these bars instead of pixels and is therefore time efficient.

In spite of the existence of line detection algorithms and the systems reported above, no research report has yet proposed to detect all classes of lines in a generic, unifying algorithm. As of now, each class of lines requires a particular detection algorithm. One possible reason for the lack of a generic approach is the fact that most researchers have done this task as a low-level stage in a drawing understanding system for a particular domain, where only a limited subset of the line types are relevant. Moreover, in the process of detecting each class of lines, almost all methods cluster all the potential constituent line segments all at once, while the line geometry and style are determined later. This blind search procedure frequently introduces inaccuracies in the grouping of the line segments, which ultimately account for inaccurate line detection. A more flexible and adaptive approach is to constantly check the line geometry and style while grouping the line segments. Indeed, this is the approach we follow in this work.

In this paper, we present the generic integrated line detection algorithm (GILDA), which detects all the major line classes that show up in engineering drawings. GILDA encompasses the identification of the two thickness values—thick and thin—the four style values—solid, dashed, dash–dotted, and dash–dot–dotted lines—and three shape values—straight, circular, and free-form curve. By combining the four line styles with the three line shapes, we define and detect a total of 12 classes of line objects that appear in engineering drawings. These classes include solid straight line (bar), solid arc (or simply, arc), solid polyline (or simply, polyline), dashed straight line, dashed arc, dashed polyline, dash–dotted straight line, dash–dotted polyline (see J1 in Fig. 1), dash–dot–dotted straight line, dash–dot–dotted arc, and dash–dot–dotted polyline.

Examining the characteristics of these line classes, we observe generic features at several levels. These are used to build the class inheritance hierarchy. The generic integrated line detection algorithm is based on this class hierarchy as well as on the generic graphic recognition algorithm, described in [21, 22]. The underlying mechanism of GILDA is a sequential stepwise recovery of line segments that meet certain thickness, style, and shape requirements. Rather than finding all the vector components of the graphic object at the same time, as is done in most current line-detection algorithms, we find for the line being detected only one new line segment that best meets the conditions constrained by the thickness, style, and shape attribute values. Before searching for the next line segment, we update the current line attribute values. This way, the current line is extended for as long as possible, while avoiding many false alarm detections.

GILDA is applied to the line fragments (bars and polylines) resulting from the sparse pixel vectorization (SPV) procedure [20]. It has been thoroughly tested as the basis of the line detection module developed within the machine drawing understanding system—MDUS [22]. SPV finds consecutively the medial axis points of a black area every several pixels by calculating the middle points of its crossing black runs. It therefore preserves the shape information (geometry and width) of lines. This shape preservation is essential for postprocessing and higher level graphics recognition, especially if no reference to the original image is assumed. The input of GILDA can be the output of any vectorization system, as long as it is a set of vectorized line segments. This set may include bars ("monolines") and polylines. Experimental results and their evaluation using the performance evaluation protocol for line detection algorithms described in [23] are also presented and discussed.

The rest of the paper is organized as follows. In Section 2 we define the 12 line classes and the class inheritance hierarchy. In Section 3 we briefly introduce the algorithmic representation of object–process diagram (OPD) [24]. We then use it to describe the generic graphic object recognition method [21, 22] and GILDA, on which it is based on. In Section 4 the syntax of the various line styles within GILDA is specified. In Section 5 we elaborate on the syntax of line shapes. In Section 6 we present additional specifications within GILDA which are required for the detection of particular classes of line objects. In Section 7 we present and evaluate experimental results, and we conclude with a discussion in Section 8.

2. LINES AND THEIR ATTRIBUTES

2.1. Line Classification

The terms defined below are used throughout the work.

(1) *Line*—a generic name of an abstract class of graphic objects in line drawings, each of which is the trace of a nonzero width pen that moves from a start point to an end point, follows a certain trajectory, which is possibly constrained by a geometric function and optionally leaves invisible segments according to some pattern.

The width of the pen is called the *line width*. The start point and the end point are called the *endpoints*. The trajectory is called the *line medial axis*. The geometric form of the line medial axis is called the *line shape*. The alternating pattern of visible and invisible segments, which is determined by the segment lengths and sequence pattern, is called the *line style*. The visible and invisible segments are called *dashes* and *gaps*, respectively.

We only consider *simple* lines, i.e., lines that do not intersect with themselves. All lines share the following common attributes.

- A line has two endpoints, which limit the extent of the line. Circles and polygons may also be considered as lines whose two endpoints coincide.
- A line has a unique, ideally constant, nonzero *width* between the two endpoints.
- A line is characterized by the *style* attribute, whose values, explained in Definitions (2)–(6), are solid, dashed, dash–dotted, or dash–dot–dotted.
- A line is characterized by the *shape* attribute, whose values are straight, circular arc, or polygonal, as listed in Definitions (7)–(9).

(2) *Solid line*—a line whose style is solid, which means that the entire line is continuously visible and traceable from end to end. In other words, it consist of a single dash and no gap between the two endpoints.

(3) *Discontinuous line*—a line whose style is not solid, that is, it consists of at least two dashes separated by one gap.

(4) *Dashed line*—a discontinuous line whose dashes are relatively equal and long, and whose gaps are relatively equal and short.

(5) *Dash–dotted line*—a discontinuous line whose dashes can be distinctly classified as long and short, alternatively. The short dashes are called *dots*. The dashes within each group are of relatively equal lengths. At least in handmade drawings, the two dashes at the two line ends are usually long.

(6) *Dash–dot–dotted line*—a discontinuous line similar to the dash–dotted line, except that every dot of the dash–dotted line is replaced with a dot–gap–dot pattern (two neighboring dots with a gap between them).

(7) *Straight line*—a line whose medial axis is constrained by

$$ax + by = c, \tag{1}$$

where (x, y) is the coordinate pair of any point on the line's medial axis, while a, b, and c are parameters. The line is limited by two endpoints $p_0(x_0, y_0)$ and $p_1(x_1, y_1)$.

(8) *Circular line*—a line whose medial axis is constrained by

$$(x - x_c)^2 + (y - y_c)^2 = r^2, \tag{2}$$

where (x, y) is any point on the line's medial axis, while (x_c, y_c) is the circular center and r is the circular radius. The line is limited by the two endpoints $p_0(x_0, y_0)$ and $p_1(x_1, y_1)$ going counterclockwise from p_0 to p_1. The two endpoints may coincide when the circular line is a full circle.

(9) *Polygonal line*—a line whose medial axis is represented by a sequence of N characteristic points p_i, where $i = 0, 1, \ldots, N-1$. The medial axis segment between every two neighboring characteristic points p_i and p_{i+1} is constrained by

$$a_i x + b_i y = c_i \quad i = 0, 1, \ldots, N-2, \tag{3}$$

where (x, y) is the coordinate pair of any point on the line's medial axis between two points $p_i(x_i, y_i)$ and $p_{i+1}(x_{i+1}, y_{i+1})$,

TABLE 2
The 12 Line Classes Obtained by Combining the Four Line Styles and Three Line Shapes

	Style			
Shape	Solid	Dashed	Dash–dotted	Dash–dot–dotted
Straight	Bar	Dashed straight line	Dash–dotted straight line	Dash–dot–dotted straight line
Circular	Arc	Dashed arc	Dash–dotted arc	Dash–dot–dotted arc
Polygonal	Polyline	Dashed polyline	Dash–dotted polyline	Dash–dot–dotted polyline

while a_i, b_i, and c_i are parameters. The entire polygonal line is limited by the two endpoints $p_0(x_0, y_0)$ and $p_{N-1}(x_{N-1}, y_{N-1})$. The polygonal line may be a closed polygon whose characteristic start and end points coincide.

A polygonal line is usually composed of a sequence of solid, equal-width lines linked end to end with optional intermediate gaps. It may also be used to approximate all line shapes other than straight and circular arc forms, including some high-order and free-form curves.

Combining the four line styles and three line shapes, we obtain the following 12 line classes, which are listed in Definitions (10)–(21) and summarized in Table 2.

(10) *Bar*—a solid straight line.

(11) *Arc*—a solid circular line.

(12) *Polyline*—a solid polygonal line consisting of a chain of equal-width bars linked end to end.

(13) *Dashed straight line*—a line whose style is dashed and whose shape is straight.

(14) *Dashed arc*—a line whose style is dashed and whose shape is circular.

(15) *Dashed polyline*—a line whose style is dashed and whose shape is polygonal.

(16) *Dash–dotted straight line*—a line whose style is dash–dotted and whose shape is straight.

(17) *Dash–dotted arc*—a line whose style is dash–dotted and whose shape is circular.

(18) *Dash–dotted polyline*—a line whose style is dash–dotted and whose shape is polygonal.

(19) *Dash–dot–dotted straight line*—a line whose style is dash–dot–dotted and whose shape is straight.

(20) *Dash–dot–dotted arc*—a line whose style is dash–dot–dotted and whose shape is circular.

(21) *Dash–dot–dotted polyline*—a line whose style is dash–dot–dotted and whose shape is polygonal.

2.2. *The Inheritance Hierarchy of Line Classes*

The object–process methodology (OPM), originally developed for information system analysis [24], has been extended to describe systems design [25] and algorithm specification [26]. It combines ideas from object-oriented approaches (OOA, e.g., [27]) and data flow diagrams (DFD, e.g., [28]) to model both the structural and procedural aspects of a system in one coherent frame of reference. The object–process diagram [24, 26], which is the graphic language of the object–process methodology, is explained in more detail in Section 3.1.

We use an OPD to graphically illustrate the class hierarchy of line objects in Fig. 2. At the root of the hierarchy is the class Line. Line is an abstract class because its style and shape attributes are not specified. Each shape of line is a lower level abstract line class, whose shape is specified but the style is not. There are three such classes: StraightLine, CircularLine, and PolygonalLine. Each line style is also represented by an abstract line class with the line style specified and the line shape unspecified. These classes are SolidLine, DashedLine, Dash–dottedLine, and Dash–dot–dottedLine. An abstract class named DiscontinuousLine is also inserted into the hierarchy as an abstraction of the three line classes that have gaps in their objects, i.e., DashedLine, Dash–dottedLine, and Dash–dot–dottedLine. Finally, the 12 concrete line classes are located at the bottom of the hierarchy. The line attributes in each concrete class are fully specified through multiple inheritance from two abstract classes, one specifying the line shape and the other specifying the line style.

To avoid the inheritance of two copies of the Line object by each one of the 12 concrete classes, as normally happens in a multiple inheritance hierarchy—one through the line shape class and the other through the line style class—we implement virtual inheritance, which is symbolized by the dotted triangle between Line and each one of its immediate specializations (inheriting classes).

3. THE GENERIC INTEGRATED LINE DETECTION ALGORITHM

Since all line classes share many common features (attributes and methods), we use these features to construct the generic integrated line detection algorithm, which is described using a set of object–process diagrams [24, 26]. To be able to freely use OPDs, we briefly explain the essential terminology, syntax, and semantics of the object–process diagrams.

3.1. *Brief Introduction to Object–Process Diagrams*

Being the graphic language of the object–process methodology [24], OPDs are graphic representations of *objects* (persistent things) and *processes* (transient things) in the universe of interest, along with the structural and procedural relationships among them. The legend in Fig. 3 shows the graphic symbols

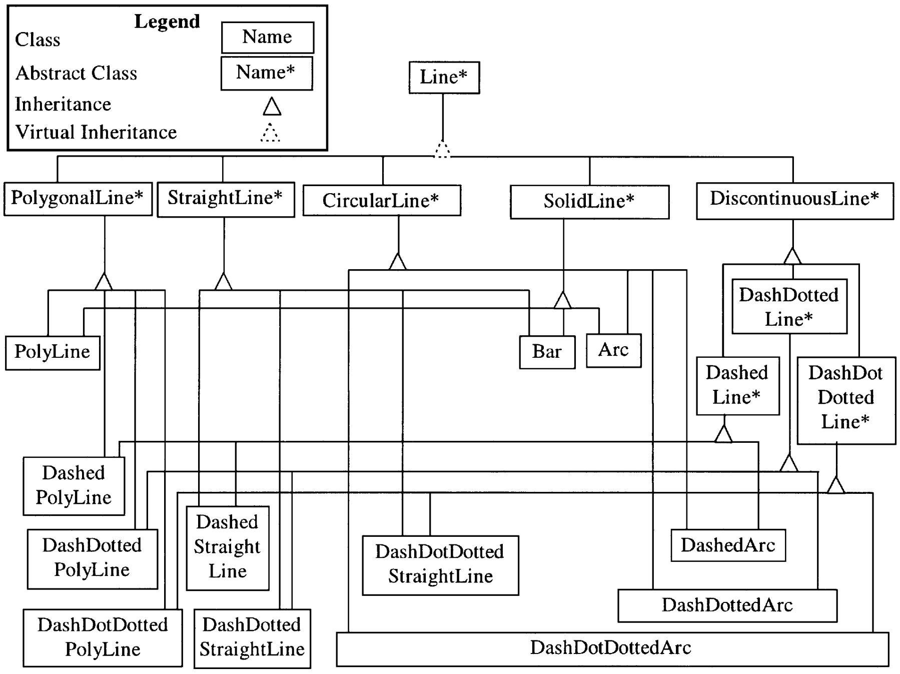

FIG. 2. OPD of the inheritance hierarchy of the classes of line objects.

discussed below. Relationships among objects (as well as among processes) are structural. They include *aggregation, generalization* (which induces *inheritance*), and *characterization* (which exists between an object and a thing that characterizes it). It therefore may occur between an object class and a process (in which case the process is referred to as a *method* or, in C++, *member function* of the class). Inheritance may also exist among processes (in which case, these processes are similar to virtual *functions* in C++ terminology).

The relationship between an object and a process is usually procedural. Procedural relations include *agent, instrument, effect, consumption,* and *result links*. An agent is an intelligent object that is involved in the process execution, such as a person or a department. An instrument is another object necessary for a process execution. An *affectee (affected object*) is an object whose content (set of attribute values) is both used by the process and changed by it. A *consumee (consumed object)* is an object that is destructed by the process, as in the C++ "delete" statement, such that it no longer exists after the process has terminated. A *resultee* (*resulting object*) is a new object constructed by the process execution, such as the object generated in the "new" C++ statement. An *owner* of a process is the unique object that owns that process. An owner may be an agent, an instrument, an affectee, or even a consumee of the process, but not a resultee. It is not mandatory for a process to have an owner. A process with no owner is called a *standalone process* and can be implemented as a function in C and C++. The procedural relation is expressed by *links* between an object and a process in OPD. A link can be thought of as a data flow as used in data flow diagrams (DFDs).

The control flow in an OPD is expressed by *control links*. A control link, depicted by a dashed arrow, may go either from a process to a process, in which case the second process is executed following the termination of the first process, or from a state of an object to a process, in which case that state invokes the execution of the process, as is the case with a branch control-flow mechanism (if–then or case statements). Control links may form a cycle, which indicates some kind of iteration.

Procedural links are always direct in the sense that there is nothing between the object and the procedurally linked process. Structural links, on the other hand, are frequently transitive and hence may be indirect. Indirect structural relationships are depicted by dotted lines. OPDs can be recursively scaleable to show different levels of details of objects and processes and their relationships. Process blowup, which is a type of upscaling or zoom in, is particularly useful for describing algorithmic processes at increasing levels of detail. The graphic symbols of the OPD terminology are shown in the object–process diagrams legends in Fig. 3. To simplify reading of the OPDs, one should first focus on the dynamic part, i.e., the processes and their input and output objects. Structural elements such as instantiation can be referred later.

3.2. *The Generic Graphic Object Recognition Approach*

Any particular class of line objects can be detected following the same pattern of the generic graphic object recognition approach [21, 22]. This approach advocates a two-step procedure, based on the hypothesize-and-test paradigm. The first step is the *hypothesis generation*, in which we assume the existence of a graphic object of the class being detected by finding its first

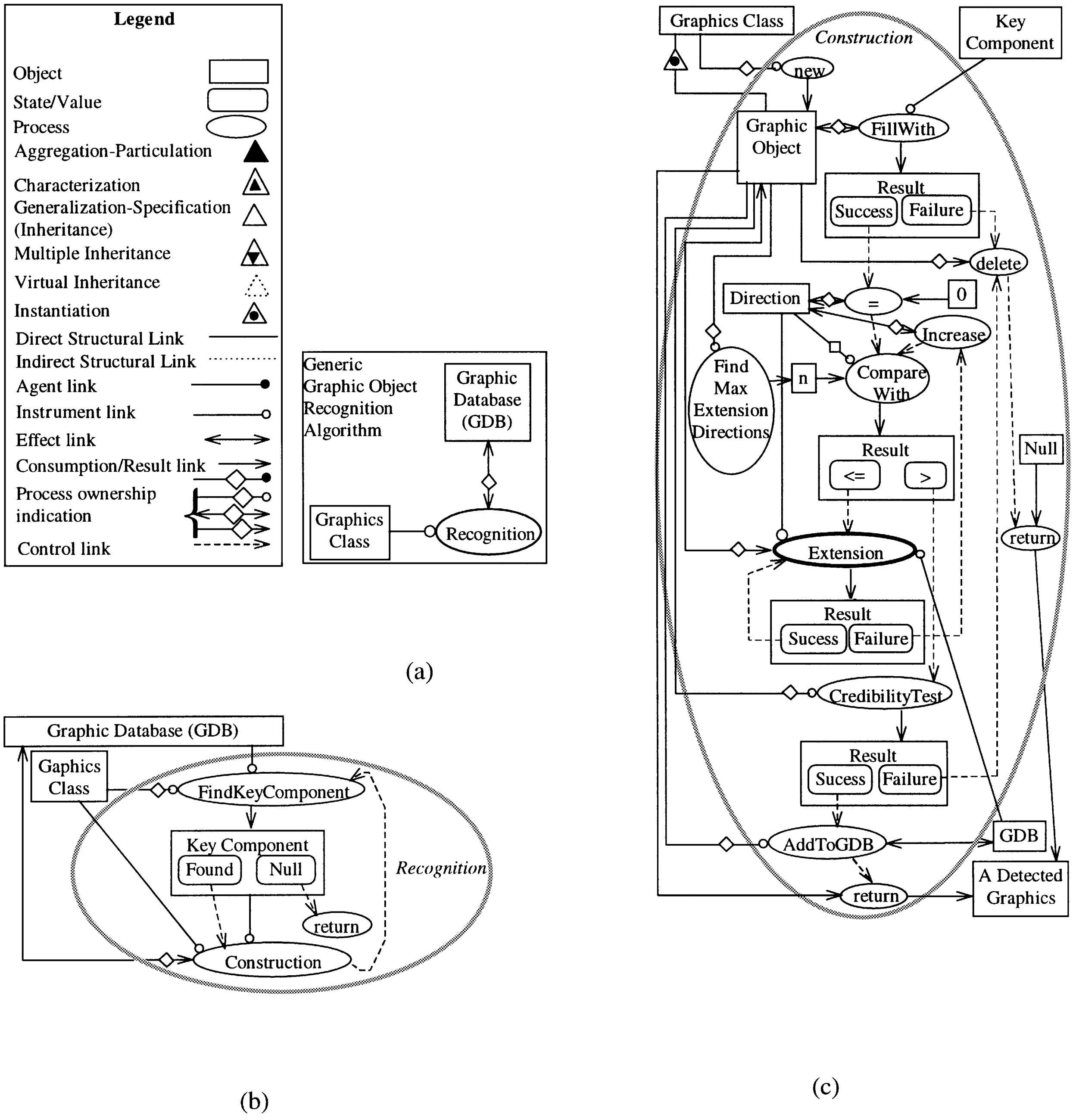

FIG. 3. OPD illustration of the Graphics Recognition Algorithm (Process): (a) top level OPD, (b) explosion of the Recognition process in (a), and (c) explosion of the Construction process in (b).

key component. The second step is the *hypothesis test*, in which we prove the presence of such graphic object by constructing it from its first key component and its other components that are detected serially. An application of the Generic Graphic Object Recognition Algorithm is shown in the top-level OPD of Fig. 3a, in which the Recognition process, owned and performed by the graphic database (GDB), takes a Graphics Class (such as one of the 12 concrete line classes in Fig. 2) as an argument (instrument) and updates GDB with the newly recognized graphic objects. Its two constituent processes are shown in the OPD of Fig. 3b, in which the Recognition process in the OPD Fig. 3a is blown up. In Fig. 3c, the Construction process of Fig. 3b is blown up to reveal the following algorithmic details. An empty Graphic Object of the Graphic Class is first created by the "new" process. By the FillWith process it is then filled with the Key Component object found by the FindKeyComponent process and transferred into the Construction process in Fig. 3b. The Graphic Object is further extended by the Extension process as far as possible in all possible directions by a stepwise recovery of its other components.

3.3. *Applying the Generic Recognition to Line Detection*

To apply the generic graphic object recognition approach to line detection, we must "instantiate" it with the appropriate line syntax and semantics. To this end, the processes of FindKeyComponent, FillWith, and IsCredible should be specified. FindMaxExtensionDirections and Extension can be specified generically at the highest level of the class Line. FindKeyComponent is the materialization of the hypothesis generation, where a first key component is selected for the hypothesized line. The details are specified in Section 6.2.

As defined in Section 2.1, all lines have exactly two endpoints, so a line can have at most two extension directions, one from each endpoint outward. The result of the FindMaxExtensionDirection process is therefore 2 for all line classes.

The details of the Extension process blown up in the OPD of Fig. 4a follows next. A Sorted Extending Candidate List is found first. Each candidate is then tested for extendibility to be joined to the current line. The extendibility test includes the processes StyleCheck and ShapeCheck. If the candidate passes the test, the current line is updated using the candidate by StyleUpdate and ShapeUpdate and the process ends successfully. If not, SequentialRetrieval takes the next candidate in the list, if there is any, to perform the same procedure. If no candidate passes the test, the process ends unsuccessfully. The Extension process is designed to apply to the Class Line, i.e., to all line classes by abstracting some of its services, as shown in Fig. 4a. The processes ShapeCheck and ShapeUpdate characterize the Line Shape Class, while the processes StyleCheck and StyleUpdate characterize the Line Style Class. The Concrete Line Class may be any one of the 12 concrete classes defined in Section 2.2. The Line Shape Class may be Straight Line, Circular Line, or Polygonal Line. The Line Style Class may be Solid Line or Discontinuous Line.

Figure 4b shows the explosion of the FindExtendingCandidate process, shown in Fig. 4a. In the process FindExtendingCandidate, an extending area, ExtendingArea, is first constructed by the GetExtendingArea process, which yields a square area stretching from the current extending point outward along the direction tangent to the line at this point. The square size is determined by the allowable gap size of the line width and style. Each one of the graphic primitives found within ExtendingArea is checked for being as extending candidate of the line currently being detected. The first process within the candidacy test is the thickness test, implemented by the process ThicknessTest. ThicknessTest succeeds if the absolute difference between the current line width and extending candidate width is less than a predefined threshold, which, in our application, is set to 2 pixels. The collinearity of the line and the candidate is then checked by the process CollinearityTest, which requires that the angle formed by the two segments be smaller than an adaptive threshold, which is 180° divided by the candidate ratio of length to width. It is normally between 15 and 60°. Finally, the ProximityTest process checks if the distance between their closest ends is no more than an adaptive threshold determined by the line width and style. The proximity threshold used in our application is twice the average gap for discontinuous lines and twice the line width for solid lines. If the graphic primitive passes all three tests, it is inserted into the Candidate List sorted by increasing distance order. In Fig. 4b, the process ThicknessTest characterizes the Line Class at the root level, while ProximityTest characterizes the Line Style Class, since only the allowable distance between two components determined by the line style is involved in the calculation, and CollinearityTest can only be specified by the Line Shape Class.

The line style and shape detection details within GILDA appear in the following two sections.

4. LINE STYLE DETECTION

4.1. *Gap Specification*

We define the gaps between two neighboring component fragments of a line as follows. For solid lines, although there should be no gap at all, gaps resulting from noise as long as twice the line width are allowed. After passing the candidacy test, two neighboring components separated by a gap are linked and the gap is filled. For discontinuous lines, the gap can be as long as twice the average gap and no shorter than twice the line width. The maximum gap length allowed by the line style is used as a parameter by the GetExtendingArea and ProximityTest processes.

4.2. *Dash and Dot Constraints*

Candidates of discontinuous line components can be either solid lines or discontinuous lines. If the candidate is a solid line, we first try to extend the candidate in the direction far away from the discontinuous line before we test it for being a new component of the discontinuous line. We try to extend it as a solid line but with the line shape of the discontinuous line being extended. By doing so, we may obtain a longer dash than the original candidate, which is broken due to noise gaps. After the candidate extension, the newly extended line is used as the candidate in the StyleCheck and ShapeCheck processes that follow. If the candidate is a discontinuous line, we do not extend it.

During the discontinuous line extension, we constantly update the average gap length and the numbers and average lengths of dashes and dots separately. If the candidate is a solid line, we first define the candidate as a dash or dot and then check if the candidate is consistent with the dash or dash–dot pattern. We require that a new dash be shorter than three times the average dash length and longer than one-third of the average dash length. If the candidate is shorter than one-third of the average dash length, it is a dot, otherwise it is a dash. This information and its length are used in the StyleCheck and StyleUpate processes. If the candidate is a discontinuous line, its average gap length, as well as the numbers and average lengths of its dashes and dots, are used in the StyleCheck process and StyleUpate processes.

As we note in Definition (5) in Section 2.1, we require that the two components at the two ends of the discontinuous line not

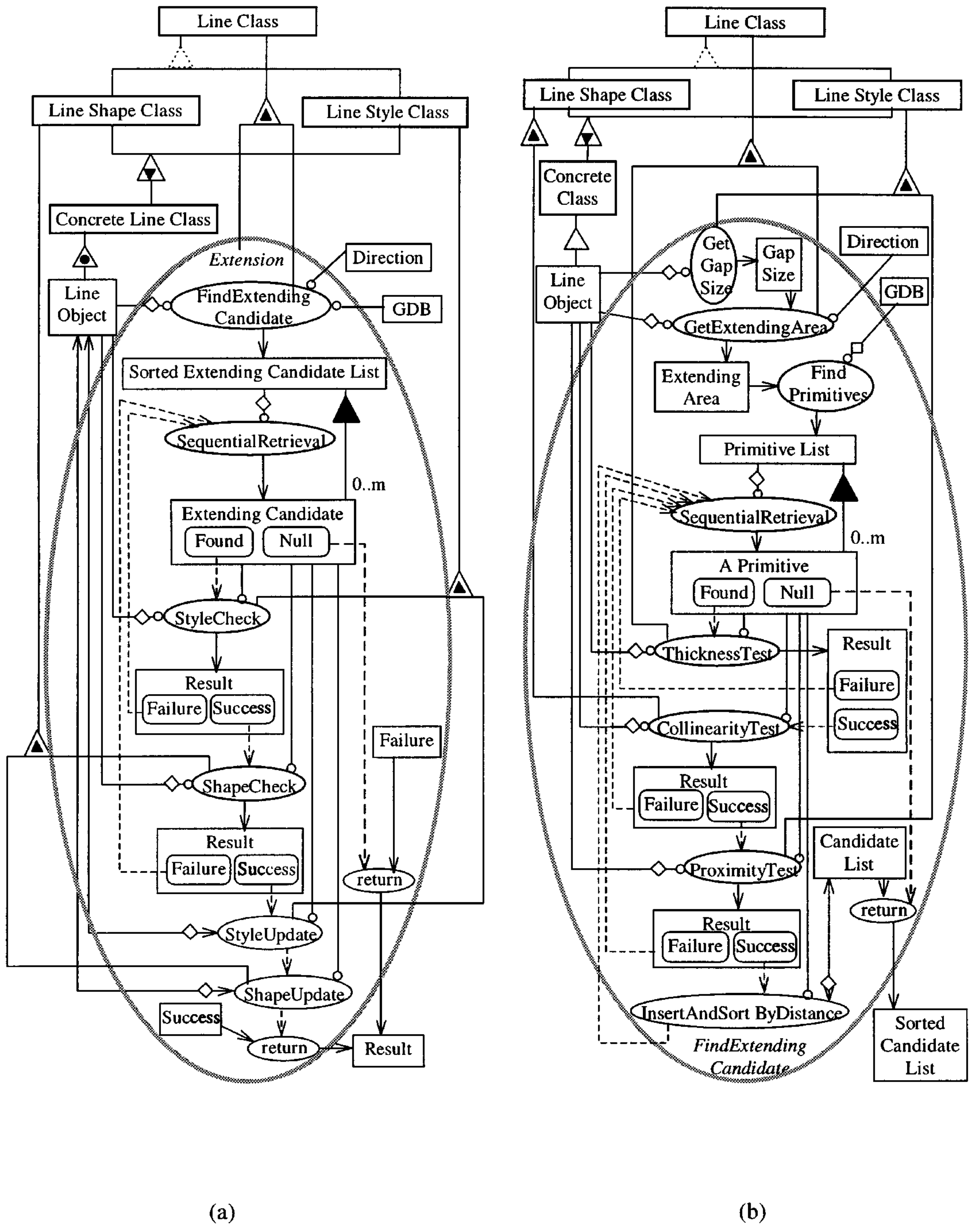

FIG. 4. (a) Blowup of Extension of Fig. 3c. (b) Blow-up of FindExtendingCandidate of Fig. 4a.

be dots but dashes, as commonly drawn. The number of dashes and the number of dots of all discontinuous line objects is then constrained by

$$N_1/(N_0 - 1) = (St - 1) \qquad (4)$$

where $St = 1$ for dashed lines, 2 for dash–dotted lines, and 3 for dash–dot–dotted lines. N_0 is the number of dashes and N_1 is the number of dots.

The dash–dot pattern is constrained as follows. Suppose we assign numbers starting from 1 to the component from one end of the discontinuous line to the other end. The first component should be a dash, the $(i * St + 1)$th component should be a dash, and the $(i * St - j)$th component should be a dot, where $i = 1, 2, \ldots, (N_0 - 1)$, j is any number that meets the condition $St - 2 < j \leq 0$.

Equation (4) and the above requirement are used in the StyleCheck process to check the dash–dot pattern. In addition,

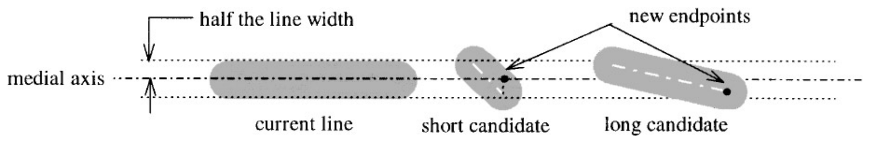

FIG. 5. Illustration of straight line extension.

StyleCheck also involves the gap check, which requires that the new gap be between half and twice the cumulative average gap.

4.3. *Discontinuous Line Parameter Update*

If the candidate passes the StyleCheck and the ShapeCheck discussed in the following section, we update its discontinuity-related line parameters as follows.

The new average gap is calculated as $(Gap + Gap_{\mathrm{av}} * (N_0 + N_1 - 1))/(N_0 + N_1)$, where *Gap* is the current gap, and Gap_{av} is the previous average gap.

The new average dash length is calculated as $(length + Dash_{\mathrm{av}} * N_0)/(N_0 + 1)$, if the new component is a dash and the new average dot length is calculated as $(length + Dot_{\mathrm{av}} * N_1)/(N_1 + 1)$, if the new component is a dot, where *length* is the length of the new component, $Dash_{\mathrm{av}}$ is the average dash length, and Dot_{av} is the average dot length of the current discontinuous line. Finally, the dash number N_0 or the dot number N_1 is incremented by 1, depending on whether the new component is a dash or a dot, respectively.

5. LINE SHAPE DETECTION

5.1. *Straight Line Specification*

Equation (1) is used in the ShapeCheck process for a straight line. In real-life drawings, lines may not be strictly straight. Therefore, we construct a strip (pipe) whose center line is the medial axis of the straight line and whose width is the line width, as shown in Fig. 5. Any straight candidate whose two endpoints fall inside this strip can be used as the new component. We take the closest candidate from the candidate list that passes the StyleCheck process as the new component.

In the ShapeUpdate process that follows straight line extension, the new endpoint of the extended line in this direction is updated as follows. If the new component is a long line, whose length is longer than three times its width ("long candidate" in Fig. 5), we use the far endpoint of the new component from the extended line as the new endpoint. If not, we do not trust the new component's endpoints. Rather, the extended line's endpoint is set as the vertical projection of the far endpoint of the new component on the medial axis of the current line, as depicted in Fig. 5 for "short candidate."

5.2. *Circular Line Specification*

The ShapeCheck and ShapeUpdate processes are extensions of the stepwise recovery arc segmentation (SRAS) algorithm [29] that apply to all the line styles we handle.

In the ShapeCheck process, a dynamic potential arc center area (PACA) is first defined, based on the current arc attributes, as shown in Fig. 6. The PACA is a rectangle whose center is the current arc center and two of its edges are parallel to the current arc's perpendicular bisector. The detailed calculation of the PACA size is given in [29]. After the PACA is determined, every point within it is checked for being a candidate of the new center. The average radius is first calculated by taking the average of the distances between the candidate—the potential arc center—and a number of characteristic points along the current arc's medial axis. Since every (solid) arc results from a polyline or a group of polylines and bars, we use the original polyline and bar characteristic points to represent the arc's characteristic points, as shown in Fig. 6. When adding a polyline segment to the arc, we use the original polyline characteristic points in the radius update calculation, as shown in Fig. 6a. When adding a bar segment to the arc, we consider the two bar endpoints, as shown in Fig. 6b. The variance of the distances from the potential center to each one of these characteristic points is calculated for each potential center within PACA. The candidate with the minimal variance is picked as the final candidate. This point has yet to pass a final test. The final test requires that (1) the difference between the average arc radius calculated for this point and the distance from this point to each one of its characteristic points must be less than half the width of the current arc, and (2) the difference between the same radius and the distance from this final candidate point to each one of the edges of the polyline formed by chaining all these characteristic points must also be

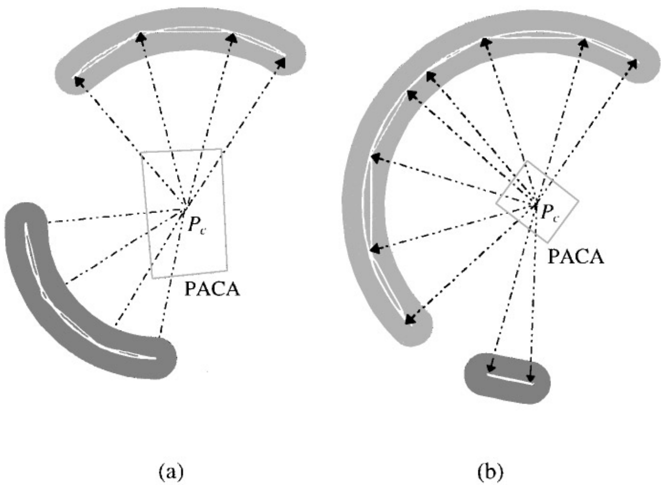

FIG. 6. Illustration of arc extension: (a) center determination when extended by an arc and (b) center determination when extended by a bar.

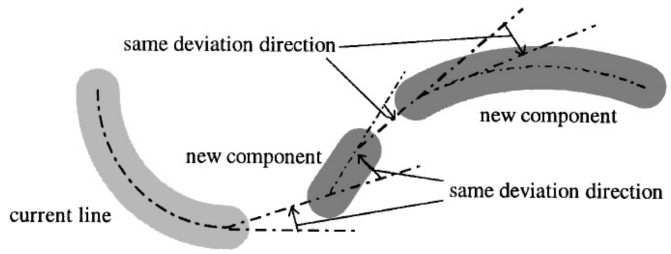

FIG. 7. Deviation direction requirement for polygonal discontinuous line extension.

less than half the width of the current arc. If the final center candidate passes this test, it passes the entire ShapeCheck. The ShapeUpdate process assigns this point as the new arc center and extends the current arc with the new component. This process also updates the arc endpoints using the farthest endpoint of the newly added component, as done when we extend a straight line. The extended arc is then checked as to whether it is a closed circle. This is done by examining the circular distance between its endpoints. If this distance passes the ProximityTest, discussed in Section 3.3, it is defined as a circle.

5.3. *Polygonal Line Specifications*

The ShapeCheck process for polygonal lines is similar to that of curved dashed line detection, discussed in [19]. For discontinuous polygonal lines, we require that the new component deviates from the gap between the current line and the new component in the same direction (either clockwise or counterclockwise) as the gap deviates from the current line, as shown in Fig. 7. For continuous polygonal lines, we require that any new component candidate be only a bar or a polyline to avoid linking arcs to polylines.

In the ShapeUpdate process, we add the original characteristic points of the new component to the characteristic point list of the current polygonal line, regardless of whether the new component is a straight, circular, or polygonal line. By so doing, we keep the detected polygonal line as the best approximation of the original polygonal line image.

6. ADDITIONAL SUBTLE REQUIREMENTS

6.1. *Line Width Update and Line Thickness Determination*

The line width is updated as the weighted sum of the current line and the new component with the weights taken as the corresponding line lengths. The line thickness attribute (whose values are "thick" and "thin") is determined by examining a line width threshold, calculated through histogramming and binarization of the widths of line segments resulting from the vectorization process [20]. If the line width is greater than the threshold, the line is thick, otherwise it is thin.

6.2. *Finding the First Key Component*

As noted in Section 3.3, the detection of any concrete line class is initialized by the FindKeyComponent process. The first key component is defined as follows. For a dashed line, the first key component can be any solid line of any length. For a dashed straight line, it can only be a bar. For a dashed arc, it can be a bar or an arc. For a dashed polyline, it can be a bar, an arc, or a polyline. For dash–dotted and dash–dot–dotted lines, the first key component can only be a dash and not a dot. This requirement is met if the dash in question is a solid line whose length is more than three times it width. The shape requirement of the first key component for the dash–dotted and dash–dot–dotted line styles is identical to that of the dashed lines, as specified above.

6.3. *The Final Credibility Test*

Since the line is obtained by a stepwise extension procedure, the cumulative error may be so large that the extended line deviates from the original requirements and violates original assumptions about shape and/or style. To avoid this kind of false alarm, a final credibility test is implemented by the CredibilityTest process, which verifies the correctness of the detected line attribute values after all the extension cycles in both directions are finished.

The CredibilityTest process includes tests of both shape and style. Only those detected liens that pass these two tests are finally accepted and used to update the GraphicDataBase. The shape test for a straight line requires that the distance between each one of the characteristic points of the line's components and the line's medial axis be less than half the line width. For circular and polygonal lines, the final shape test does nothing and lets all lines pass.

The CredibilityTest also includes a style test, which checks the minimal number of both components and patterns. The minimal number of components is 2 for dashed lines, 3 for dash–dotted lines, and 4 for dash–dot–dotted lines. For a solid line, although it is unnecessary to require a minimal number of components, we do require that at least two original components be involved, otherwise we can simply keep each one of the original components as a solid line. The final pattern test uses Eq. (4) as the criterion and gives a tolerance of 20%. We also require that the average dash length be at least twice the average dot length, otherwise the dots may be considered as dashes of a dashed line.

7. EXPERIMENTAL RESULTS AND EVALUATION

We have implemented the GILDA as the basis of the line detection module within the machine drawing understanding system [22], which is developed in C++ on **SGI Indy** and **Indigo2** workstations (**IRIX5.3**) and **SUN** Sparcstations (**Solaris2.5**). The executable codes of these two versions are available from the addresses in [30]. The detection order of the line classes is shown in the OPD in Fig. 8. The preprocessing is a sparse pixel vectorization procedure [29], which yields vector fragments—bars and polylines—from the original scanned raster file for further processing in the line detection process.

We have tested the GILDA with about 50 real-world drawings and obtained very good results, as the following examples show.

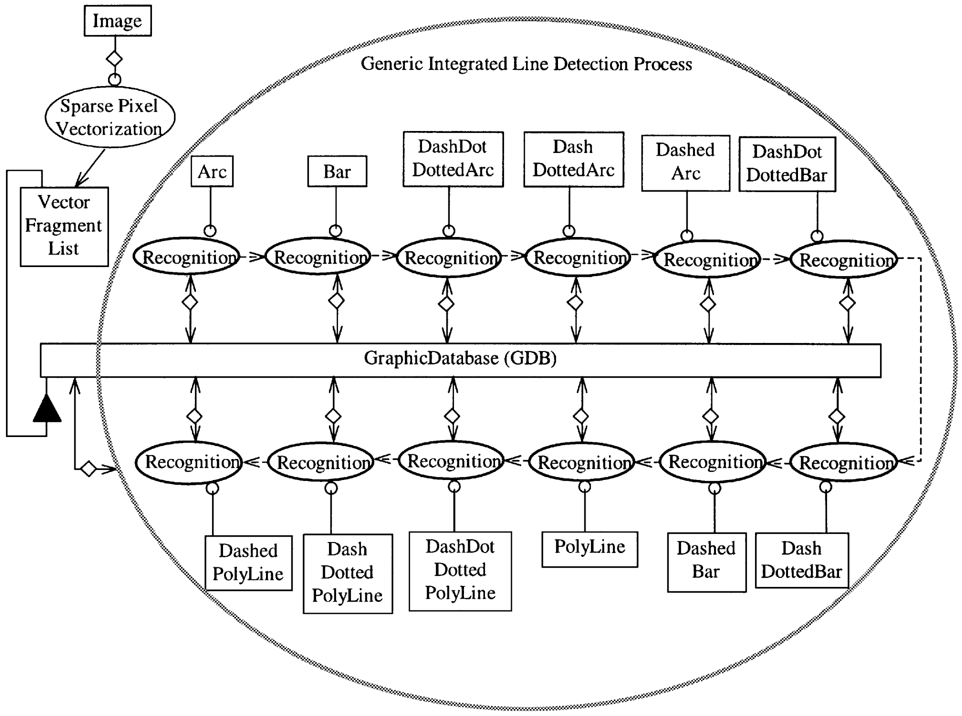

FIG. 8. OPD of the line detection order.

Figure 9 is the line detection results of (a) SPV and (b) GILDA applied on the drawing image presented in Fig. 1. All the detected lines in Fig. 9b are displayed using a single line width and solid line style. As we can see from Fig. 1, the drawing is noisy due to paper quality and scanning processing. The line edges are not smooth. There is even a broken dash segment in the dash–dot–dotted line at the bottom. It is also very complex due to the presence of not only almost all line classes but also a complex mixture of line classes, e.g., lines that are tangent to arcs, lines which intersect at very small angles, and discontinuous lines which pass through hatched areas. However, as we can see from Fig. 9b, almost all arcs (including circles) and discontinuous lines have been correctly detected. In addition, as indicated inside Fig. 9b, some discontinuous lines are broken and some detected arcs are false alarms due to the image quality and complexity. The two dashed lines across two hatched areas (the vertical one at the left hand and the horizontal one in the middle of the drawing) are also detected correctly. The dash–dot–dotted line at the bottom of the drawing is also fully detected despite the break in one dash. The failure of arc segmentation is that the open angle is so small that SPV produces only a bar on the arc image. The common arc false alarms are due to the polyline shape formed by several short bars.

Figure 10a is an ANSI [3] drawing with many solid and dashed circles. The intermediate result of sparse pixel vectorization (SPV) is shown in Fig. 10b. The result of line detection by the GILDA is displayed in Fig. 10c in solid lines with a single line width. The drawing is also complex, since there are many concentric circles and small open angle arcs. The bigger dashed circle consists of only short bar dash segments and is intercepted by many other lines at small angles. However, all solid arcs and circles are correctly detected, but several arcs are false alarms. Three out of the four small dashed circles are correctly detected. The fourth small dashed circle at bottom right is not detected because its top left dash is too long. Even the biggest dashed circle outside the biggest solid circle is correctly detected, though it is broken in two parts. The top part is longer than 3/4 circle and the bottom part consists of three dashes. This is because part of an arc dash participates in a false arc (see Fig. 10c, bottom right) and the environment is highly cluttered. Failure of finding one dash may cause the extension to halt in the current direction. All eight straight slanted short dashed lines are also correctly detected. The two dash–dotted lines marking the centers for the central concentric circles are also detected correctly, while the four dash–dotted lines marking the four small hole centers are detected as several broken dash–dotted lines, because the extension fails to span the long gap caused by the intersection of the biggest thick arc at a small angle. Another detected dashed line (vertical at the right bottom) is a false alarm caused by joining the thick bar with a tail of a leader (arrow) at the bottom right of the drawing. Occasionally, the dashed line false alarms may appear due to the dashed patterns formed accidentally by several irrelevant line segments.

Figure 11 is a clean drawing taken from ISO [1] with solid and dash–dotted circles, and straight dash–dotted lines, which we used to test the GILDA. Figure 11a is the original image and Fig. 11b is the arc segmentation results, in which all the dashes of the dash–dotted circles are detected as arcs. Figure 11c is

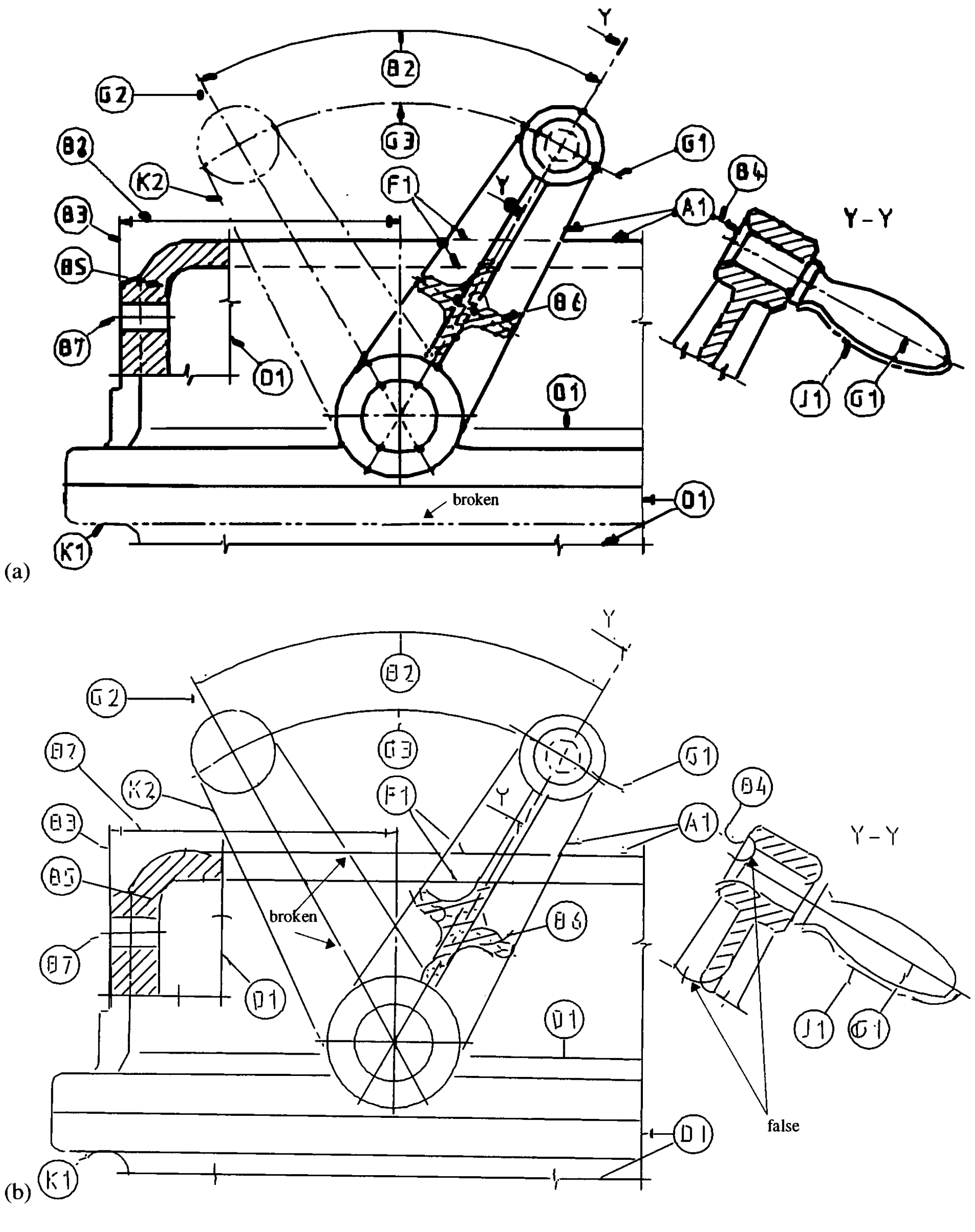

FIG. 9. Line detection results by (a) SPV and (b) GILDA on the drawing image in Fig. 1.

the screen display of the final line detection results, where the dark gray represents solid circles, while the light gray represent detected dash–dotted bars and circles. All lines in this drawing are correctly detected, except for the upper and lower dash–dotted bars, which are shorter than those in the original drawing because one dash of each is broken near the thick solid circle already at the preprocessing stage.

We have also tested the GILDA for discontinuous and polygonal line detection. The test image in Fig. 12a is manually prepared using Microsoft Paintbrush software. Its sparse pixel vectorization (SPV) result is shown in Fig. 12b. The result of GILDA is shown in Figs. 12c and 12d. In Fig. 12c, the detected polygonal discontinuous lines are displayed with solid style and a single line width. The discontinuous free from curves are very nicely detected and only two of them are broken in one location each. Figure 12d shows the detection results dumped from the screen, where each discontinuous pattern is represented by a different gray level.

We have automatically evaluated the GILDA on separate tasks in [23] using the Vector Recovery Index (VRI), which is a single measure of the accuracy of the detected lines compared with their ground truths. It is an objective and comprehensive index that involves the detection accuracy of the width, shape, style, characteristic points, and the detection fragmentation and combination. The arc detection capability of the GILDA depends on the width and the open angle. For $\pi/8$ arcs with too thin (e.g., 1

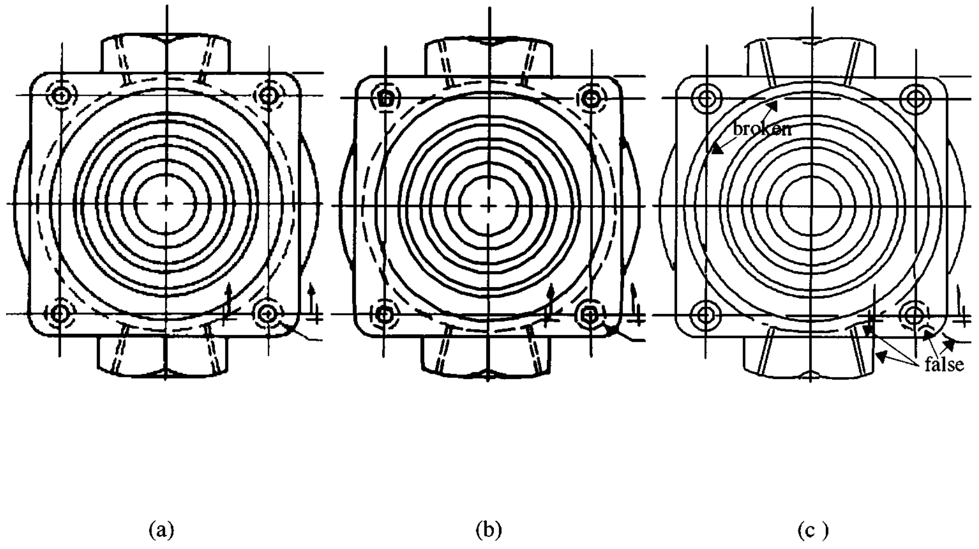

FIG. 10. Line detection results by GILDA on the an ANSI drawing: (a) image, (b) sparse pixel vectorization result, and (c) detected lines.

or 2 pixels) or too thick (i.e., when the distance from the chord to the arc is less than half the width), the VRI can be as low as around 0.40. For circles, the VRI can be as high as 0.97, which is an exceptionally good result. The bar detection can also be as high as more than 0.90 for clear and simple drawings [23]. We have manually measured the vector ground truths of the image in Fig. 11a and used it in the performance evaluation of the GILDA. The ground truth lines and their detection evaluation are listed in Table 3. The detected lines and their evaluation are listed in Table 4. The overall Detection Rate (D_v) [23] for the entire drawing, which results from the detection quality of all the ground truth lines, is 0.81 and the overall False Alarm Rate (F_v) [23], which results from all the detected lines, is 0.17. The VRI (which is the average of D_v and $1-F_v$) we obtained is 0.82, which is acceptable [23]. However, this value is a little bit smaller than what we had expected since the result is quite good as evaluated by human vision. From Tables 3 and 4, we can see the quality of each line entities. For example, the detection quality of the ground truth 12 (which is the lowest tangent dash–dotted line) is only 0.43. This is mainly because it is not fully detected as a whole dash–dotted line since it is matched with detected lines 3 and 14. Even a well-detected dash–dotted line as judged by human vision—ground truth line 8 and detected line 13, which is the central horizontal dash–dotted line, obtains a detection quality value of only 0.76. This is mainly due to the fact that the detected endpoints deviate from the ground truth. Human vision cannot measure this small error. Another possible reason is that the ground truths we measured are not precise, and this is the very reason that we cannot trust the manually measured ground truth for Figs. 9 and 10, which are quite complex. Hence,

TABLE 3
The Ground Truth Lines in Fig. 11a and Their Detection Evaluation

Ground truth	Width (pixels)	Shape	Style	End point1	End point2	Arc center	Arc radius	Matched detected line	D_v
0	9	Circle	Solid			288,238	204	4	0.91
1	3	Circle	Solid			288,238	163	6	1.00
2	8	Circle	Solid			288,238	37	8	0.82
3	9	Circle	Solid			867,248	149	5	0.91
4	3	Circle	Solid			867,248	108	7	0.77
5	8	Circle	Solid			867,248	36	9	0.74
6	3	Circle	Dash–dotted			288,238	185	15	0.87
7	3	Circle	Dash–dotted			867,248	129	16	0.71
8	3	Straight	Dash–dotted	61,233	1040,250			13	0.76
9	3	Straight	Dash–dotted	292,12	284,466			10	0.77
10	3	Straight	Dash–dotted	869,98	864,418			12	0.77
11	3	Straight	Dash–dotted	292,55	868,120			0,1,11	0.75
12	3	Straight	Dash–dotted	285,425	865,378			3,14	0.43
Total									0.81

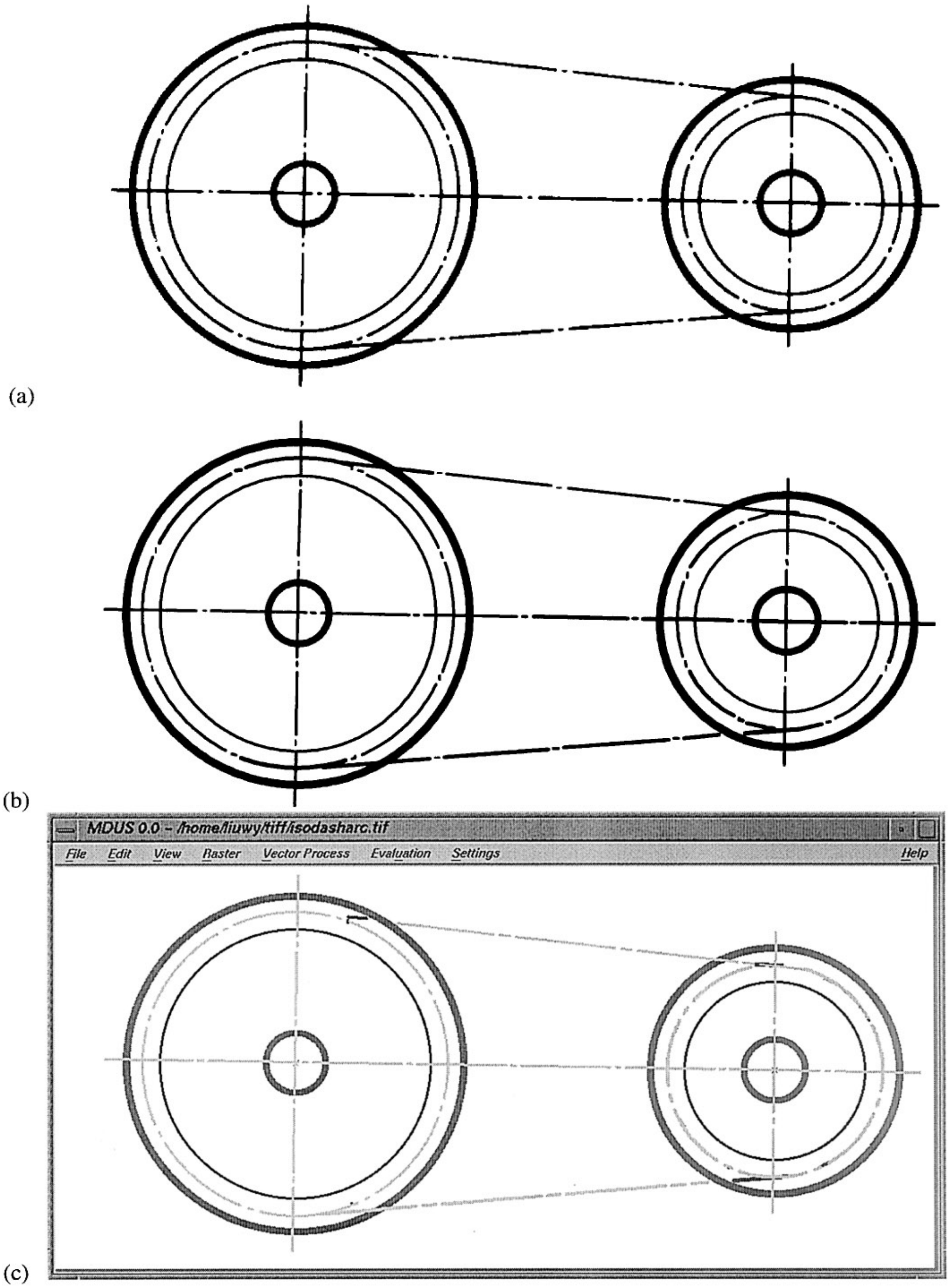

FIG. 11. GILDA performance on a real life ISO drawing: (a) image, (b) arc segmentation, and (c) discontinuous line detection.

their automated evaluation using the protocol in [23] is not available.

The time performance of the GILDA including the SPV preprocessing is listed in Table 5. For drawings with average size of 1000 × 1000 pixels the processing is up to 30 s on SGI Indigo2. The is quite fast compared to most of the currently available systems. The time efficiency is also due to the stepwise extension, which avoids blind search in the entire drawing. The Planar Position Index (PPI) [21] data structure, which indexes all line entities using their planar positions and therefore facilitates the area search for lines, also contributes to the efficiency of GILDA.

8. CONCLUSION

A generic and integrated line detection algorithm, based on the generic graphics recognition approach is developed,

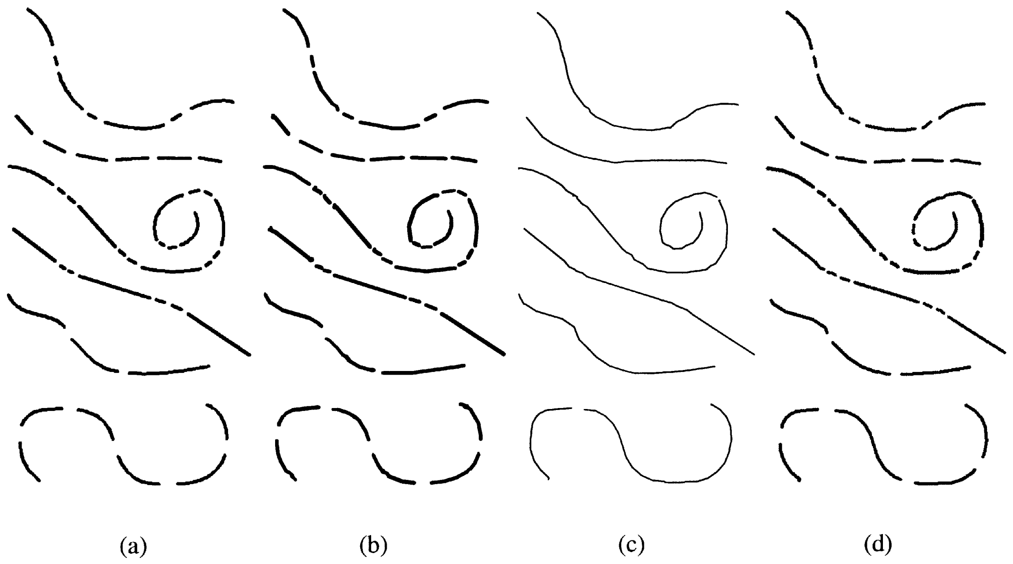

FIG. 12. MDUS performance on a real life ISO drawing: (a) image, (b) sparse pixel vectorization, (c) arc segmentation, and (d) dashed line detection.

implemented, and evaluated. The algorithm abstracts the recognition as a stepwise recovery of components of the graphic objects. We define 12 classes of lines that appear in engineering drawings and use them to construct a class inheritance hierarchy which abstracts the line features used in the algorithm. The line classification in the paper uses parameters that are more relaxed than those defined by drawing standards (e.g., ISO [1]) so as to be adapted to human vision perceptions and line classification. These standards define line types (thickness and styles) but does not refer to shapes (geometry). The classification in this paper does not affect line type definitions in any standard or software package (e.g., AutoCAD). It abstracts those definitions by adding geometry factors to allow for refined recognition which will be used at higher levels of understanding. All line classes are stepwise extended to both ends. In each extension cycle, only one new component, which best meets the current line's shape and style constraints is joined to the line and the line is extended using it. The details of the algorithm are presented at increasing levels of detail by a set of object–process diagrams, which correspond to the actual C++

TABLE 4
The Detected Lines in Fig. 11c and Their Evaluation

Detected line	Width (pixels)	Shape	Style	End point1	End point2	Arc center	Arc radius	Matched ground truth	F_v
0	4	Straight	Solid	352,60	374,62			11	0.63
1	5	Straight	Solid	844,118	875,119			11	0.65
2	3	Straight	Solid	355,407	355,408				1.00
3	5	Straight	Solid	817,380	881,375			12	0.74
4	9	Circle	Solid			288,238	203	0	0.09
5	9	Circle	Solid			866,247	149	3	0.09
6	3	Circle	Solid			288,238	163	1	0.00
7	3	Circle	Solid			866,247	108	4	0.23
8	8	Circle	Solid			288,237	36	2	0.18
9	8	Circle	Solid			866,246	37	5	0.28
10	3	Straight	Dash–dotted	291,11	283,466			9	0.24
11	3	Straight	Dash–dotted	411,68	903,124			11	0.07
12	3	Straight	Dash–dotted	868,79	863,418			10	0.28
13	3	Straight	Dash–dotted	60,232	1038,249			8	0.23
14	4	Straight	Dash–dotted	317,420	786,382			12	0.45
15	4	Arc	Dash–dotted	351,64	351,63	288,238	185	6	0.13
16	4	Arc	Dash–dotted	903,124	843,121	866,247	128	7	0.23
Total									0.17

TABLE 5
Time Performance of the GILDA (on SGI Indigo2)

Figure	Size (pixels)	Time (s)
9	1232 × 810	28
10	824 × 600	11
11	1056 × 468	6
12	400 × 800	1

code that implements the algorithm. The OPD set is instrumental in clarifying the data and control flow of the algorithm. We use a rather complex inheritance hierarchy to abstract as much as possible the line features in the algorithm. As we show in the experiments, the algorithm demonstrates high performance on a variety of real-life and synthetic drawings. It is robust in some cases, such as line mixture, intersection, tangency, and touching letters, due to the stepwise recovery procedure. It is also time efficient due to avoiding blind search. The generalization of line detection for all line shapes and styles is a very significant achievement, as it provides for a single unifying algorithm that is easy to maintain and improve, following state-of-the-art software engineering principles of genericity and reuse.

However, GILDA also produces false alarms, especially for solid arcs, where several bars form polylines, as in some cases in Figs. 9 and 10. Some solid arc false alarms may be avoided if the threshold (half the line width) in the shape check is set to be more strict (e.g., one quarter of the line width). But this would also decrease the detection rate. The trade-off between detection and false alarms rates is a subject for further research. An alternative is to introduce higher level knowledge from higher level recognition and understanding and feedback. This is another research subject. Occasionally, dashed line false alarms may appear due to the dashed patterns formed accidentally by several independent line segments. This could also be avoided at higher levels of recognition. It should be noted that the lines detected by GILDA are separate individual entities. GILDA is not designed to integrate or group them into higher level entities, such as a dashed rectangle or a dimension set. These higher level graphic objects will be detected at higher level understanding phases, where more domain-specific syntax and semantics are introduced.

REFERENCES

1. *Technical Drawings*, ISO Standard Book 12, First ed., 1982.
2. N. Sidheswar, P. Kannaiah, and V. V. S. Sastry, *Machine Drawing*, Tata/McGraw–Hill, New Delhi, 1980.
3. *Dimensioning and Tolerancing*, ANSI Y14.5M, 1982.
4. *Mechanical Drawings*, Mechanical Industrial Press, Beijing, China, 1985. [In Chinese]
5. C. S. Fahn, J. F. Wang, and J. Y. Lee, A topology-based component extractor for understanding electronic circuit diagrams, *Comput. Vision Graphics Image Process.* **44**, 1988, 119–138.
6. R. Kasturi, S. T. Bow, W. El-Masri, J. Shah, J. R. Gattiker, and U. B. Mokate, A system for interpretation of line drawings, *IEEE Trans. Pattern Anal. Machine Intell.* **12**(10), 1990, 978–992.
7. V. Nagasamy and N. A. Langrana, Engineering drawing processing and vectorization system, *Comput. Vision Graphics Image Process.* **49**(3), 1990, 379–397.
8. L. Boatto *et al.*, An interpretation system for land register maps, *IEEE Comput.* **25**(7), 1992, 25–32.
9. P. Vaxiviere and K. Tombre, Celesstin: CAD conversion of mechanical drawings, *IEEE Comput.* **25**(7), 1992, 46–54.
10. S. H. Joseph and T. P. Pridmore, Knowledge-directed interpretation of mechanical engineering drawings, *IEEE Trans. Pattern Anal. Machine Intell.* **14**(9), 1992, 928–940.
11. V. Poulain d'Andecy, J. Camillerapp, and I. Leplumey, Kalman filtering for segment detection: Application to music scores analysis, in *Proc. of the 12th International Conference on Pattern Recognition, Jerusalem, Isreal, 1994*, Vol. I, pp. 301–305.
12. D. H. Ballard, Generalizing the Hough transform to detect arbitrary shapes, *Pattern Recognit.* **13**(2), 1981, 111–122.
13. P. L. Rosin and G. A. West, Segmentation of edges into lines and arcs, *Image Vision Comput.* **7**(2), 1989, 109–114.
14. D. Dori, Vector-based arc segmentation in the machine drawing understanding system environment, *IEEE Trans. Pattern Anal. Machine Intell.* **17**(11), 1995, 1057–1068.
15. D. Pao, H. F. Li, and R. Jayakumar, Graphic feature extraction for automatic conversion of engineering line drawings, in *Proc. 1st International Conference on Document Analysis and Recognition, Saint-Malo, France, 1991*, pp. 533–541.
16. C. P. Lai, and R. Kasturi, Detection of dashed lines in engineering drawings and maps, in *Proc. 1st International Conference on Document analysis and Recognition, Saint-Malo, France, 1991*, pp. 507–515.
17. Y. Chen, N. A. Langrana, and A. K. Das, Perfecting vectorized mechanical drawings, *Comput. Vision Image Understanding* **63**(2), 1996, 273–286.
18. G. Agam, H. Luo, and I. Dinstein, Morphological approach for dashed lines detection, in *Graphics Recognition—Methods and Application* (R. Kasturi and K. Tombre, Eds.), Lecture Notes in Computer Science, Vol. 1072, pp. 92–105, Springer, Berlin, 1996.
19. D. Dori, W. Liu, and M. Peleg, How to win a dashed line detection contest, *Graphics Recognition—Methods and Application* (R. Kasturi and K. Tombre, Eds.), Lecture Notes in Computer Science, Vol. 1072, pp. 286–300, Springer, Berlin, 1996.
20. W. Liu and D. Dori, Sparse pixel tracking: A fast vectorization algorithm applied to engineering drawings, in *Proc. of the 13th International Conference on Pattern Recognition, Vienna, Austria, 1996*, Vol. III (Robotics and Applications), pp. 808–811.
21. W. Liu, D. Dori, L. Tang, and Z. Tang, Object recognition in engineering drawings using planar indexing, in *Proc. of the First International Workshop on Graphics Recognition, Pennsylvania State University, 1995*, pp. 53–61.
22. W. Liu and D. Dori, Automated CAD conversion with the machine drawing understanding system, in *Proc. of 2nd IAPR Workshop on Document Analysis Systems, Malvern, PA, Oct. 1996*, pp. 241–259.
23. W. Liu and D. Dori, A protocol for performance evaluation of line detection algorithms, *Machine Vision Appl.* **9**(5), 1997, 240–250. [Special Issue on Performance Characterisitics of Vision Algorithms]
24. D. Dori, Object–process analysis: Maintaining the balance between system structure and behavior, *J. Logic Comput.* **5**(2), 1995, 227–249.

25. D. Dori and M. Goodman, From object–process analysis to object–process design, *Ann. Software Eng.* **9**, 1996, 1–25.

26. W. Liu, and D. Dori, Extending object–process diagrams for the implementation phase, in *Proc. of the Third International Workshop on the Next Generation of Information Techniques and Systems, Nave Ilan, Israel, June 30–July 7, 1997*, pp. 207–214.

27. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

28. T. De Marco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

29. D. Dori and W. Liu, Arc segmentation from complex line environments—A vector-based stepwise recovery algorithm, in *Proc. of the Fourth International Conference on Document Analysis and Recognition, Ulm, Germany Aug. 1997*, pp. 76–80.

30. http://iew3.technion.ac.il:8080/∼liuwy/liuwy_pro.shtml or ftp.technion.ac.il/pub/supported/ie/dori/MDUS/sgimdus.gz and sunmdus.gz.