Network-Based Recognition of Architectural Symbols

Christian Ah-Soon and Karl Tombre

LORIA–CNRS–INRIA, B.P. 239, 54506 Vandœuvre-lès-Nancy CEDEX, France

Abstract. In this paper, we propose a method for recognizing architectural symbols. The method is based on the description of the model through a set of constraints on geometrical features, and on propagating these constraints through a network, following an idea first proposed by Messmer and Bunke. One of the advantages of this approach is the possibility to incrementally build and update the model, when new symbols have to be taken into account.

1 Introduction

Our team has been working for more than two years on the analysis of architectural drawings. The ultimate aim is to reconstruct a 3D model of a building from the analysis of design-phase drawings. For this purpose, we have developed two complementary methods for reconstructing the geometric model of a level from its vectorization [1]. But both of these methods rely heavily on a correct recognition of architectural symbols. These symbols are much less normalized than those which can be found in other technical domains. We therefore need a flexible method, capable of easily integrating new symbol models with minimal computation overhead in the recognition phase.

After explaining our two main sources of inspiration for this work (§ 2), we describe our model (§ 3) and the way it is used for recognition (§ 4), before explaining how new symbols can be added to the model (§ 5).

2 State of the Art

The recognition of graphical symbols is a well-known problem, for which many methods have been proposed [2]. A first family of methods has been adapted to documents such as diagrams, basically made of symbols and connecting lines (see [9] as a recent example). Other applications include cartography [7] and printed music scores [5]. But there have been few attempts at recognizing architectural symbols; one of the only works we are aware of is that of Lladós *et al.* [3], who use attributed graph matching to recognize symbols taken from a set of known models. Their method has proven to be efficient, even for hand-drawn drawings. But it necessarily suffers from the usual limitations of graph matching methods.

In our quest for a good recognition method, we felt the need for *flexibility* and *genericity*. As architectural drafting is much less normalized than other technical domains, we come upon large variations in the way basic elements such as doors or windows are represented. We therefore *cannot* build an *a priori* set of models and decide that these are

the only symbols we will recognize. We must be able to incrementally add new models to the knowledge base, with minimal computational overhead during recognition.

A first system which inspired us was that of Pasternak [6]. In his ADIK kernel system, he uses graphical specifications of the symbols, based on a number of predicates and on constraints between parts of the same geometric composition object. Object recognition is activated through a triggering mechanism. The whole knowledge base is represented as a structural/geometric taxonomy.

Keeping the main idea of a constraint-based, hierarchical modelling, we turned to another method for more efficient management of the set of models. Messmer and Bunke [4] proposed a method which allows for model pre-compilation through the use of a network, where all model descriptions are gathered at once; the features are the input to this network and "trickle down" until one of the terminal nodes—i.e. one of the model symbols—is activated. This work was based on graph isomorphism; in our case, as we use constraint propagation, we have adapted the network concept to these constraints.

3 Symbol Modelling

To explain our model, we will assume in this article that the vectorization of the document image yields a set of segments. This can easily be extended to segments and arcs. Let $\mathcal{F}$ be the set of *features*, i.e. n-uples of distinct segments; let $\mathcal{P}$ be the set of *predicates*, i.e. boolean functions. The size of a feature is defined as the number of its segments, and the size of a predicate is the number of its arguments. Let $\mathcal{C}$ be the set of *constraints*, defined as $\{c \in \langle \mathcal{P}\ pr, \mathcal{F}\ fe\rangle | (size(fe(c)) \geq 1) \wedge (size(fe(c)) = size(pr(c)))\}$. Thus, a constraint is made of a predicate and of a feature, and it applies to the segments of this feature. We define the size of the constraint as being the size of its feature.

Let $\mathcal{D}$ be a set of descriptions. A description is defined by a feature, whose segments represent the model symbol, and by a set of constraints. These constraints apply to the segments of the feature, and are of two kinds: connection constraints, which describe connection relations between segments, and simple constraints. We define the size of a description as being the size of its feature.

For instance, we can define the description of a lozenge (Fig. 1) as follows:
$d_{sl} = \langle \{sd_1, sd_2, sd_3, sd_4\}, \{cc_1, cc_2, cc_3, cc_4, cs_1, cs_2, cs_3, cs_4\}\rangle$
$cc_1 = \langle (sd_1, sd_2), p_c\rangle$ $cc_2 = \langle (sd_2, sd_3), p_c\rangle$ $p_c : x \times y \mapsto point1(x) = point2(y)$
$cc_3 = \langle (sd_3, sd_4), p_c\rangle$ $cc_4 = \langle (sd_4, sd_1), p_c\rangle$
$cs_1 = \langle (sd_1, sd_2), p_s\rangle$ $cs_2 = \langle (sd_2, sd_3), p_s\rangle$ $p_s : x \times y \mapsto length(x) = length(y)$
$cs_3 = \langle (sd_3, sd_4), p_s\rangle$ $cs_4 = \langle (sd_4, sd_1), p_s\rangle$

4 Symbol Recognition

4.1 Use of a Network

Although Messmer and Bunke use a different matching mechanism, subgraph isomorphism, we adapted several of their ideas to our method. For instance, we use a network

to model the descriptions and thus perform the search for all possible symbols at once, instead of trying separately to match a candidate with all possible models. For this, we search separately for all the features verifying each constraint, and we then merge these features to get the symbol.

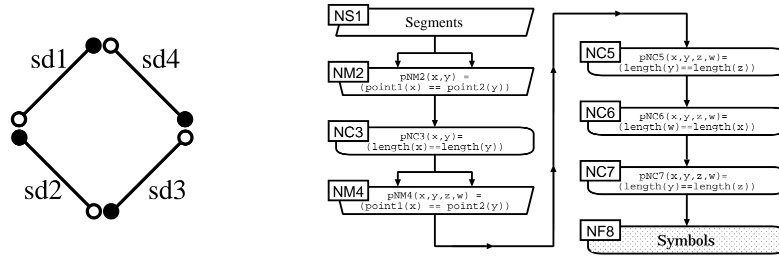


Fig. 1. Network for recognition of a lozenge.

Thus, in order to detect lozenges of description d_{sl} (§ 3) among the segments of an image, we can use an 8-node network (Fig. 1). By using the various predicates (p_{NM2}, $p_{NC3}, p_{NM4} \dots$), we get the constraints which describe the lozenge. Thus, the features which end up in NF8 represent lozenges.

4.2 Generalization of the Network

The search for symbols works through propagation of the segments (yielded by vectorization) through a network. This network is made of four kinds of nodes : NNSegment, NNMerge, NNCondition and NNFinal. These nodes are connected through father–son links; each node can have at most two fathers, but can have several sons. Each node tests some constraints, and can thus be seen as a "filter", which only transmits to its sons the features (sets of segments) which verify the tested constraints. These features, created only once by each node, can be used by all the sons of the node. At the end, the segments of the features which have "trickled" down to the terminal nodes of the network represent the corresponding symbols.

For each network, there is only one **NNSegment** node, which corresponds to the root of the network. This node initializes the recognition process, creates a one-segment feature for each segment, and sends it to all its sons (Alg. 1). A **NNCondition** node has only one father. It tests the constraint on the features sent to it by its father. If the constraint is satisfied, the NNCondition node propagates the feature to its sons (Alg. 2). A **NNMerge** node has two father nodes, and gathers the features sent by its fathers, if they verify a connection constraint. The resulting feature, if any, is sent to the sons of the NNMerge node (Alg. 3). Note that in order to allow the NNMerge nodes to gather all their fathers' features, each node in the network has to keep a local storage of the features it is transmitting. The **NNFinal** nodes are the terminal nodes; they have one father and no sons. Each of these nodes corresponds to one of the symbols which have to be recognized. When a feature reaches such a node, it has went through a number of

NNMerge and NNCondition nodes and has verified their constraints. To get the actual symbol, it is therefore sufficient to get the set of features stored in the NNFinal node.

Alg. 1 NNSeg.transmission(image I)

for all segments s of I **do**
  newFea ← create a feature from s
  myFeatures.add (newFea)
  for all my sons n **do**
    n.transmission (newFea, me)
  end for
end for

Alg. 2 NNCond.transmission($\mathcal{F}$ f, $\mathcal{N}$ p)

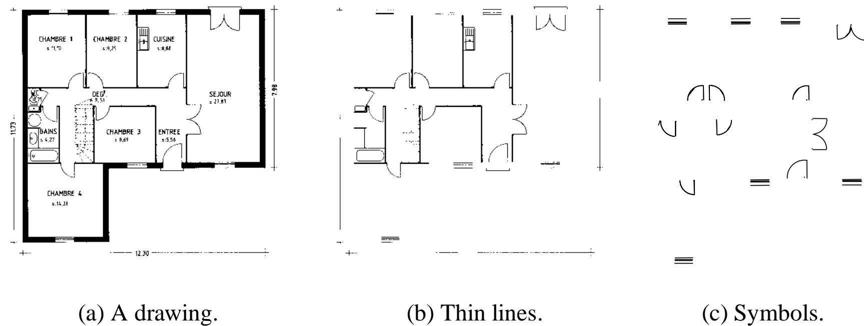
if f verifies myPredicate **then**
  myFeatures.add (f)
  for all my sons n **do**
    n.transmission (f, me)
  end for
end if

Alg. 3 NNMerge.transmission($\mathcal{F}$ f, $\mathcal{N}$ p)

for all features g disjoint of f and sent by my other father (not p) **do**
  if p = myFather1 **then**
    newFea ← merge (f, g)
  else
    newFea ← merge (g, f)
  end if
  if newFea verifies myPredicate **then**
    myFeatures.add (newFea)
    for all my sons n **do**
      n.transmission (newFea, me)
    end for
  end if
end for

4.3 Use of the Network

Two kinds of features can be extracted by low-level techniques: segments and arcs. As vectorization and arc recognition methods are always noisy, the resulting set of graphical entities may contain extraneous segments, especially at the junctions of thick lines. Actually, we are looking for symbols such as windows and doors, which are always represented with thin lines; we therefore separate thick lines from thin lines, using simple mathematical morphology. The thin lines image is vectorized independently of the thick lines image (Fig. 2(b)) [8]. This reduces the number of artefacts in the set of vectors.



(a) A drawing. (b) Thin lines. (c) Symbols.

Fig. 2. Result for a simple drawing (1700×1600) vectorized in about 300 segments and arcs.

We have tested our network on eleven architectural drawings such as the one represented in Fig. 2(a), with nine descriptions of doors and windows. With a computation time of 5 to 30 seconds on a SUN Sparc Ultra 1, the network recognizes most of the represented symbols (Fig. 2(c)). In the following table, we give for each drawing the number of symbols to be recognized (S), the number of symbols recognized by the system (R) and the number of false hits (F). Most of the latter stem from redundant recognition (e.g. double doors being also recognized as two simple doors).

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**
S	14	11	15	12	12	15	14	14	16	14	15
R	13	11	14	11	10	15	14	14	14	13	14
F	1	0	3	2	0	0	1	4	1	4	2

5 Building the Network

For a set of descriptions, it is of course possible to build several networks, each relative to one description. But as we want to accelerate the symbol recognition process, our aim is to factorize as much as possible the constraints which are common to several symbols, and to find the most efficient ordering in a common network. For this, we use several heuristics. After constructing the root node NNSegment, we proceed incrementally and sequentially: the descriptions are ordered by increasing number of constraints, and added to the network one after the other. For each new symbol, we only add nodes for constraints which are not already tested in the network.

5.1 Constraints Which Are Already Tested

One of the strengths of this approach is the ability to use constraints common to several descriptions. When a new description is added to the network, we look for constraints in this description which the network can already test.

For this, we input the segments $sd_1, \dots, sd_t$ of the description to the network (t being the size of the description). When they propagate through the network, these segments will be checked by all the constraints already available there, and this yields features which correspond to one of the constraints of the description. This can be done by slightly modified versions of the algorithms used in the recognition phase: Alg. 4, 5 and 6 are very similar to the previous Alg. 1, 2 and 3.

After this propagation, the network contains several features, localized in all the nodes through which the segments verifying the description have been propagated. For instance, let us look at the following d_{new} description:

$d_{new} = \langle \{sd_1, sd_2, sd_3, sd_4\}, \{cc_1, cc_2, cc_3, cs_1\} \rangle$

$cc_1 = \langle (sd_1, sd_2), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$

$cc_2 = \langle (sd_2, sd_3), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$

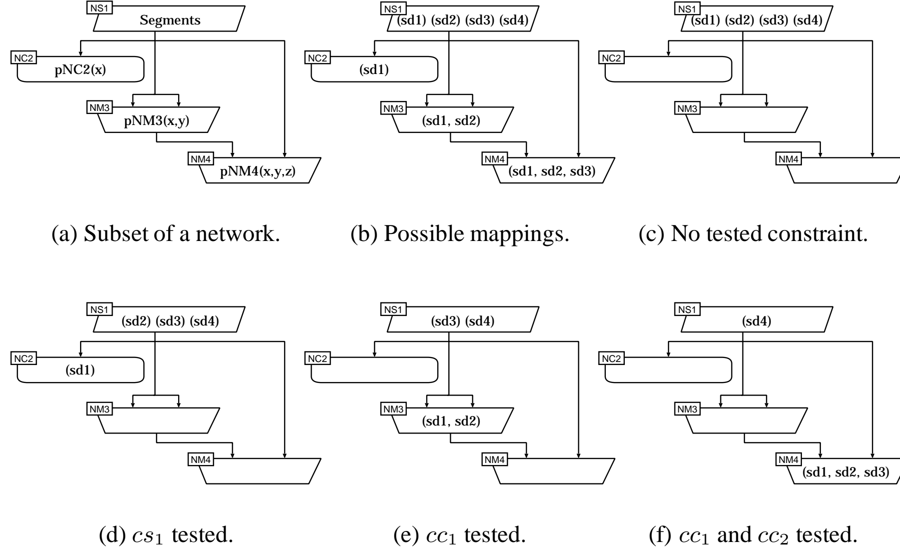
$cc_3 = \langle (sd_1, sd_4), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$

$cs_1 = \langle (sd_1), p_{cs1} : x \mapsto length(x) \leq 20 \rangle$

which we want to add to an existing network (Fig. 3(a)), where:

$p_{NC2} : x \mapsto length(x) \leq 20$

$p_{NM3} : x \times y \mapsto point1(x) = point2(y)$



(a) Subset of a network. (b) Possible mappings. (c) No tested constraint.

(d) cs_1 tested. (e) cc_1 tested. (f) cc_1 and cc_2 tested.

Fig. 3. Looking for constraints already tested by the network.

$$p_{NM4} : x \times y \times z \mapsto point1(y) = point2(z)$$

After transmission of the model segments in the network, using the previously described algorithms, the network contains seven features (Fig. 3(b)).

Alg. 4 NNSeg.transD($\mathcal{D}$ d)

for k ← 1 to size (d) **do**
  feat ← create one feature from sd_{k}
  myFeatures.add (feat)
  for all my sons n **do**
    n.transD (feat, me, d)
  end for
end for

Alg. 5 NNCond.transD($\mathcal{F}$ f, $\mathcal{N}$ p, $\mathcal{D}$ d)

c2 ← create one constraint from f and myPredicate
if there is a constraint c of d such that c2 is an extension of c (§ *5.5*) **then**
  myFeatures.add (f)
  for all my sons n **do**
    n.transD (f, me, d)
  end for
end if

Alg. 6 NNMerge.transD($\mathcal{F}$ f, $\mathcal{N}$ p, $\mathcal{D}$ d)

for all features g disjoint of f and filtered by my other father (not p) **do**
  if p = myFather1 **then**
    newFeat ← merge (f, g)
  else
    newFeat ← merge (g, f)
  end if
  c2 ← create one constraint from newFeat and myPredicate
  if there is a constraint c of d such that c2 is an extension of c (§ *5.5*) **then**
    myFeatures.add (newFeat)
    for all my sons n **do**
      n.transD (newFeat, me, d)
    end for
  end if
end for

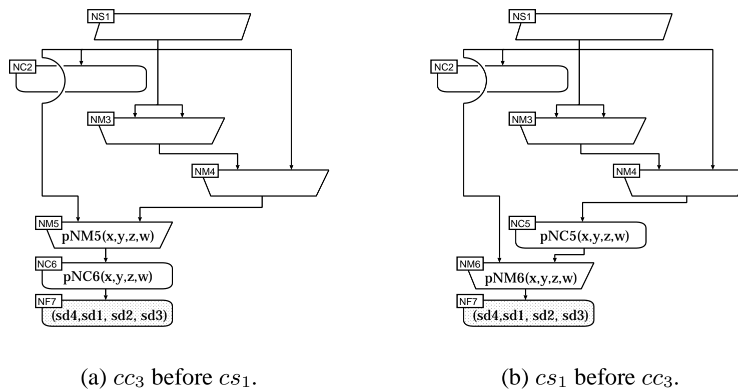
5.2 Disjoint Features Set

Among the features present in the network after propagation (§ 5.1), let us choose a set of disjoint features, i.e. a set such that each segment of the model is present in one and only one feature. Generally, several choices are possible for such a set. All these choices are valid for the incrementation of the network. But it is better to choose the set which yields the most compact network. We therefore try to maximize the number of constraints already tested by the traversed nodes.

In the previous example with d_{new}, it is possible to create four disjoint features sets (Fig. 3(c), 3(d), 3(e) and 3(f)), but we choose the last set, for which two constraints are already tested by the network.

5.3 Adding Simple Constraints

After having chosen a set of disjoint features, we have to decide how to add the new nodes to the network. This depends on the order in which the remaining constraints have entered the network (Fig. 4). We decided to process the simple constraints sequentially, starting with those of smallest size (e.g. a constraint on only one segment will be processed before a constraint on several segments). This relies on the fact that the smallest constraints are supposed to have most discriminating power, and thus it is interesting to find them at an early stage in the network.



(a) cc_3 before cs_1.　　(b) cs_1 before cc_3.

Fig. 4. Two choices when adding the remaining constraints of d_{new}, starting from Fig. 3(f).

Before we create the node which will check the simple constraint, we must group the corresponding segments into a common feature. If they are not already grouped, i.e. if they are spread into different features, we use the `mergeFeatures` function (§ 5.4) to create the appropriate NNMerge nodes. When all segments concerned by the simple constraint are grouped, the corresponding NNCondition node can be added to the sons of the node where the last feature is found. The feature can then be removed from the

latter node and added to the newly created node. Every time a new simple constraint is added to the network, we also check whether this new constraint can be used to verify other untested constraints of the description, to minimize the size of the network and improve its performances.

Finally, when all simple constraints have been added, the remaining connection constraints, which have not already been taken into account, are added with the `mergeFeatures` function (§ 5.4). As all segments of the description are related to each other through constraints, and as all constraints have been inserted in the network, all segments of the description are included in a common feature. The corresponding NNFinal node, which represents the new symbol to be recognized, can therefore be created and added to the sons of the node containing this feature.

5.4 Adding Connection Constraints

There are several cases where we need to merge features: to create a unique feature when adding a simple constraint, or to add the remaining connection constraints in a description (§ 5.3). The algorithm we use (Alg. 7) to add the nodes merging these features takes as arguments the list of features to be merged, the network where they are located, and the list of constraints between pairs of segments belonging to the features to be merged. The features are merged two by two, and this results in the creation of the corresponding NNMerge nodes. If no connection constraint is found, the node is created with a $true$ predicate.

When all the features have been merged into a single final feature, the remaining connection constraints, if any, related to two segments of this feature, are added to the network as NNCondition nodes.

For example, the merging of the three features located in three nodes (Fig. 5(a)), through the following connection constraints:
$cc_1 = \langle (sd_5, sd_6), p_{cc1} : x \times y \mapsto (point1(x) = point2(y)) \rangle$
$cc_2 = \langle (sd_1, sd_2), p_{cc2} : x \times y \mapsto (point2(x) = point1(y)) \rangle$
$cc_3 = \langle (sd_4, sd_5), p_{cc3} : x \times y \mapsto (point2(x) = point2(y)) \rangle$
leads to the creation of a NNMerge node for the cc_1 and cc_2 constraints (Figs. 5(b) and 5(c)), and to the creation of a NNCondition node for the cc_3 constraint (Fig. 5(d)).

Alg. 7 mergeFeatures (list$\langle \mathcal{F} \rangle$ feat, network r, list$\langle \mathcal{C} \rangle$ const)

while size(feat) > 1 **do**
  f1 ← smallest feature of feat
  f2 ← smallest feature of feat not equal to f1, such that there is a constraint cc in const between 2 segments from f1 and f2
  if no such feature f2 exists **then**
    f2 ← smallest feature of feat not equal to f1
    p ← true
  else
    p ← predicate which tests cc
    remove cc from const
  end if
  newNode ← create NNMerge from p and the nodes of r which contain f1 and f2

add newNode to r
remove f1 and f2 from r and from feat
add the feature resulting from the merge of f1 and f2 to feat and to newNode
end while
while there are constraints cc left in const **do**
add cc to a new NNCondition node, as son of the last new node
end while

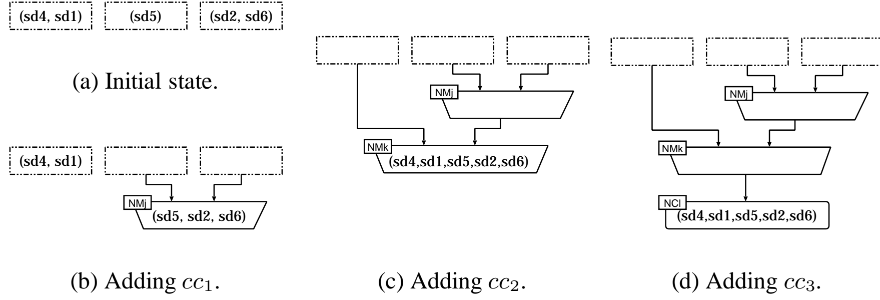


Fig. 5. Merging different features by using connection constraints.

5.5 Constraint Checking

Generally, the predicate tested by a NNMerge node or a NNCondition node is not equal to the predicate of the corresponding constraint. This stems from the fact that we can only check that the segments to which the constraint is related are *included* in the features received by the node, which can also contain other segments, or contain the right segments in an other order than that expressed by the constraint. To take into account these variations, we generalize the predicates which are put into the nodes when we create the network.

Let $size$ be the recursive function defined by:

$$size : n \mapsto \begin{cases} 1 \text{ if } n \text{ is of type NNSegment} \\ size(father(n)) \text{ if } n \text{ is of type NNFinal or NNCondition} \\ size(father_1(n)) + size(father_2(n)) \text{ if } n \text{ is of type NNMerge} \end{cases}$$

The size of a NNMerge and NNCondition node, defined by this function, also corresponds to the size of the tested predicate and that of the feature which can be sent by the node.

Let d be the constraint that the node to be created must verify, and let m be the feature from which we create the node. If we create a NNCondition node, it is the feature coming from the father node of this node. If we create a NNMerge node, it is the union of the features coming from the two fathers. By definition, this feature m contains the segments to which constraint d refers. The predicate of d must be a

restriction of the predicate p which we must add to the new node, modulus a change in the order of its arguments. We say that constraint c defined by $\langle m, p\rangle$ is an extension of d for m. The injection l defined on $[1, size(d)] \rightarrow [1, size(c)]$ by:

$$\begin{cases} \forall i \in [1, size(d)], n_i = m_{l(i)} \\ \forall x \in \mathcal{S}^{size(c)}, p(x_1, ..., x_{size(c)}) = q(x_{l(1)}, ..., x_{l(size(d))}) \end{cases}$$

gives the order of the arguments for the two predicates, as it returns the location of the segments of d's feature in c's feature. For a feature m containing all segments of the d's feature, the injection l is defined uniquely. Actually, there is only one constraint c, having m as its feature, for which c is an extension of d. For example, for the feature (sd_2, sd_1, sd_3, sd_4), $c = \langle(sd_2, sd_1, sd_3, sd_4), p : x \times y \times z \times w \mapsto length(z) = 2.length(y)\rangle$ is an extension of $d = \langle(sd_3, sd_1), q : x{\times}y \mapsto length(x) = 2.length(y)\rangle$.

6 Conclusion

We have presented an adaptation of Messmer and Bunke's network approach to constraint propagation. Our first results are hopeful. We are currently working on improving the low-level processing, for a better input to the system. We also have to evaluate the performances of our system on a larger number of drawings, with a larger set of model symbols.

References

1. C. Ah-Soon and K. Tombre. Variations on the Analysis of Architectural Drawings. In *Proceedings of Fourth International Conference on Document Analysis and Recognition, Ulm (Germany)*, pages 347–351, August 1997.
2. Atul K. Chhabra. Graphic Symbol Recognition: An Overview. In Karl Tombre and Atul K. Chhabra, editors, *Graphics Recognition—Algorithms and Systems*, volume 1389 of *Lecture Notes in Computer Science*, pages 68–79. Springer-Verlag, April 1998.
3. J. Lladós, J. López-Krahe, and E. Martí. A system to understand hand-drawn floor plans using subgraph isomorphism and Hough transform. *Machine Vision and Applications*, 10(3):150–158, 1997.
4. B. T. Messmer and H. Bunke. Automatic Learning and Recognition of Graphical Symbols in Engineering Drawings. In R. Kasturi and K. Tombre, editors, *Graphics Recognition—Methods and Applications*, volume 1072 of *Lecture Notes in Computer Science*, pages 123–134. Springer-Verlag, May 1996.
5. H. Miyao and Y. Nakano. Note Symbol Extraction for Printed Piano Scores Using Neural Networks. *IEICE Transactions on Information and Systems*, E79-D(5):548–554, May 1996.
6. B. Pasternak. *Adaptierbares Kernsystem zur Interpretation von Zeichnungen*. Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.), Universität Hamburg, April 1996.
7. H. Samet and A. Soffer. MARCO: MAp Retrieval by COntent. *IEEE Transactions on PAMI*, 18(8):783–798, August 1996.
8. K. Tombre, C. Ah-Soon, Ph. Dosch, A. Habed, and G. Masini. Stable, Robust and Off-the-Shelf Methods for Graphics Recognition. In *Proceedings of 14th International Conference on Pattern Recognition, Brisbane, Australia*, August 1998.
9. Y. Yu, A. Samal, and S. C. Seth. A System for Recognizing a Large Class of Engineering Drawings. *IEEE Transactions on PAMI*, 19(8):868–890, August 1997.