University of Alberta

THE APPLICATION OF INDUCTIVE LOGIC PROGRAMMING TO ASSIST DIRECTORY LAYOUT DESIGN

by

Ping Gu ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 1999

0-612-40055-7

[illegible]

Ping Gu
1316 Carling Ave, #2208
Ottawa, Ontario
Canada, K1Z 7L1

Date: Dec. 8. 1998

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The Application of Inductive Logic Programming to Assist Directory Layout Design** submitted by Ping Gu in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . . . [illegible] . . . .
Dr. Randy G. Goebel

. . [illegible] . . . .
Dr. Renée Elio

. Russell Greiner . . . . . .
Dr. Russell Greiner

. [illegible] . . . .
Dr. Alinda Friedman

Date: . Dec 8/98.

To my parents: D.B. Gu and Xiuqin

Abstract

Today's increasing dynamics of commercial Yellow Pages^{TM} (YP) telephone publishing market requires effective tools that support easily customized Yellow Pages^{TM} layouts and allow frequent updates. Yellow Pages^{TM} Pagination System (YPPS) which adopts an AI approach based on constraints in the problem of automated Yellow Pages^{TM} layout provides a flexible and intuitive formulation of complex layout restrictions. Hence, the system is easier to maintain, extend and customize. However, one difficulty arises when a customer needs to learn the complex layout constraints specification language in order to specify his preferences. Therefore, we study the problem of how to obtain layout constraints from a group of layout examples.

The specification of the layout rule induction problem can be subdivided into two parts: one is to parse the relationships between different types of YP objects; the other is how to represent the relationships and use them to produce layout rules. We study layout rules from a real world application and define a set of relations which capture the attributes of real-world layout rules. We implement a prototype system which can analyze those relations from layout examples and generate files for Progol. Progol is then used to generalize layout rules. The test results show that Progol is able to construct those known layout rules given good and bad examples. Also, it can induce some new and potentially useful rules if provided some good and bad examples of how to place a particular object type.

Acknowledgements

I would like to express my sincere gratitude towards Dr. Randy G. Goebel, my supervisor, for his guidance and on going support. His continuous encouragement and advice throughout the course of my work was invaluable. I would also like to thank my committee members: Dr. Russell Greiner, Dr. Renée Elio, and Dr. Alinda Friedman for their careful reading and constructive criticisms. Also, I should thank my fellow graduate students and staff of the Department of Computing Science for creating a superb work environment.

Finally, I wish to thank to my husband, Hao Luan, and my family for all their encouragement and undying confidence.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 Preliminary

With increases in the volume and dynamics of information distributed by multimedia, pagination and page layout problems have recently been paid more attention. The problem of Yellow Pages^{TM} (YP) Pagination and layout (YPPL) is one of finding a compact and harmonious positioning of text and advertisements on the pages of a commercial telephone directory ([1]). In practice, it is treated as a multi-dimensional placement problem and hence is a complex computational task. In this dissertation, we refer Yellow Pages^{TM} to YP.

Figure 1.1 provides an example of a typical spread (double-page). We use this figure to define some terminology. YP objects are classified by headings. For instance, auto parts and auto rentals in Figure 1.1 are two specific headings. *Display ads* (DAs) can span more than one column horizontally and the corresponding *incolumn object* (ICs) which contain the text streams are of one-column width. A *heading* (HD) consists of a set of DAs and ICs in lexicographic order. Any space remaining after placement of DAs and text are placed is filled with *filler* material. The filler material can be further subdivided into *display filler* (DF) and *incolumn filler* (IF).

Of previously reported approaches, the three most relevant for general pagination and page-layout tasks are those that use rule-based methods, optimization by simulated annealing ([1]) and constraint-based methods.

The rule-based approach has been applied to the YPPL problem and its newspaper

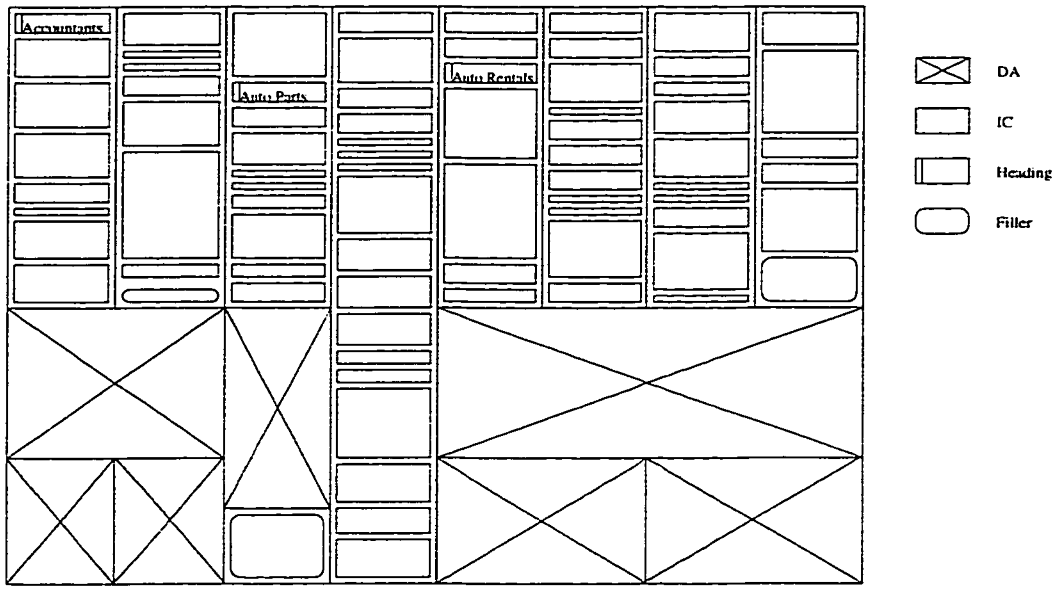


Figure 1.1: **An example of a typical YP directory**

equivalent with moderate success ([2]). Although "if-then" rules are helpful in defining an objective function operationally, it appears that it is unsuitable for fully automated YP layout, which is a very challenging combinatorial optimization problem. Those "if-then" rules play an assisting role in YP design.

A simple stochastic search technique based on simulated annealing is presented in [1], and can generate extremely compact and harmonious page layout. It formulates a YPPL problem as an optimization problem in which the task is to position display ads and incolumn text stream segments on sequential pages so as to minimize total page length, and satisfy page-format requirements and positional relations between DAs and ICs. The algorithm has been applied to sample data from a real telephone-directory, and produces solutions that are significantly shorter and better than the published ones. However, since the approach uses general criteria for evaluating solutions, it cannot satisfy particular requirements from different users. In this case, it is not easily adapted for a general automated layout problem, which may occur in different media and may have different requirements.

Constraint programming is broadly applicable to many computationally intensive tasks. [3] presents a constraint-based pagination tool for YP telephone directories, called *YPPS* (Yellow Pages$^{\text{TM}}$ Pagination System). In YPPS, the YP problem is treated as a constraint satisfaction problem. Basically, layout requirements are stated by means of constraints. Expertise about the application domain can be exploited by providing heuristics which guide the constraint solving process. By using some powerful constraint processing techniques, such as constraint propagation over finite domains and dedicated constraint abstractions, the system applies a coherent collection of layout constraints to a stream of YP objects to achieve aesthetically appealing and rationally organized layouts.

The constraint-based methods provide a flexible formulation of complex layout conditions. The hope is that the constraint processing technology is flexible enough to help transform YP publishing to the volatile electronic publishing market.

1.1.2 The YPPS

YPPS is the first system that integrates constraint programming and artificial intelligence methods for the problem of layout for telephone book Yellow Pages^{TM}. Most current pagination algorithms exploit heuristics that are procedurally encoded for computational efficiency, with little support to flexibly accommodate user-specified preferences. The AI approach based on constraints provides a flexible and intuitive formulation of complex layout restrictions, as well as a declarative representation of design knowledge([3]). Hence, a constraint-based layout system is easier to maintain, extend, and customize. Moreover, it is easier to adapt the system to various presentation media and to tailor to a specific user profile by setting design parameters.

Figure 1.2 illustrates the schematic architecture of YPPS. The input includes a set of multimedia items (YP objects) and a set of pagination parameters, such as the user's layout preference (e.g., spacing, weighted grid), output media (e.g., paper, HTML, CD), and resource limitations (e.g., space, time). Both the system designer and end users can select a set of layout constraints from a pre-defined constraint library in YPPS according to company-, application-, and user-specific criteria. Moreover, the end user can select among a set of placement strategies and optimization criteria (e.g. filler minimization vs. balancing of DAs and ICs), which are mapped onto constraints or propagators. The overall architecture shows the powerful flexibility of the system.

One important feature of YPPS is its underlying constraint language $YellowPages^{TM}$

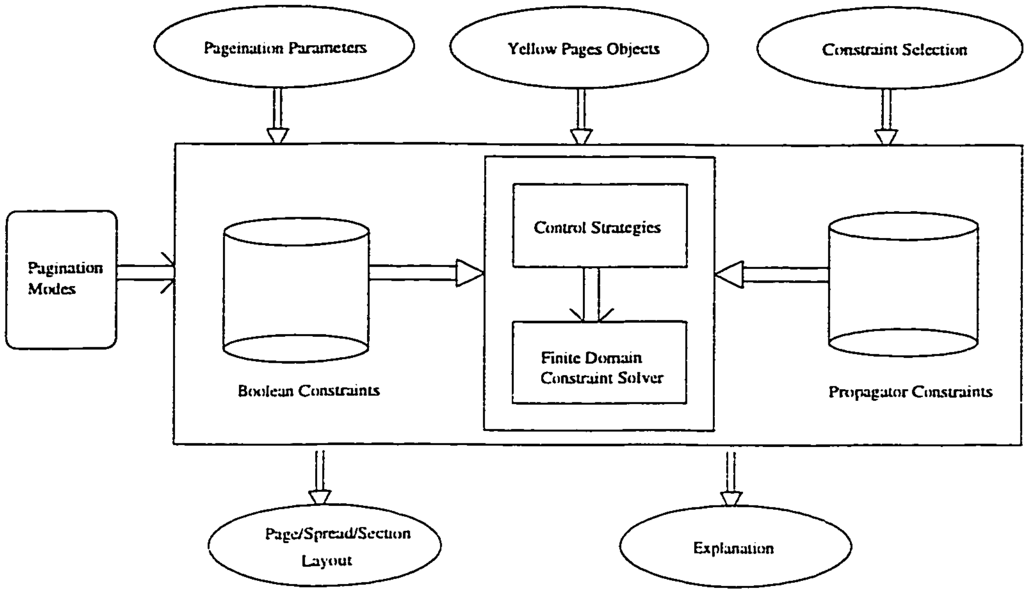


Figure 1.2: **The architecture of the YPPS system**

Language (YPL). YPL is a declarative, concise, and powerful constraint language which supports a relatively "natural" specification of layout characteristics by the page layout designer. It helps free the end user from cumbersome procedural layout algorithms and makes it easier to formally represent the layout knowledge of pagination experts.

1.2 The Problem

Today's increasing dynamics of commercial YP telephone publishing market requires effective tools that supports easily customized YP layouts and allow frequent updates. YPPS is flexible in that it supports both system designers and the end user in specifying constraints in a high-level language. However, it is difficult for a client to understand the constraint representation language. Therefore, the problem of how to represent layout constraints based on the client's requirements is important.

It is usually easy for a client to provide a group of layouts which reflect his layout preference and some general layout characteristics. Layout examples are typically divided into two classes: good examples and bad examples. A good example shows a layout preference in a positive way. On the other hand, a bad example illustrates what is wrong with regard to a specific layout requirement. From the layout examples, a layout designer can induce the valid constraints and use them in a YP pagination system. The problem is to specify a set of layout rules from the examples which clients provide. It is a challenge for a person to accomplish the task. With the state-of-the-art computer-aided induction techniques, we can take layout examples as input and generalize layout rules which can assist the layout designer in constructing a set of application-specific constraints, then YPPS can use the constraints to automatically produce aesthetically pleasing layouts which satisfy client requirements.

1.3 Overview of Approach

1.3.1 Inductive Logic Programming

Inductive learning is one form of machine learning. Various logical formalisms have been used in inductive learning systems to represent examples and concept descriptions. In general, these formalisms range from propositional logic to full first-order predicate calculus. We distinguish two classes of learning systems based on its formalism for representing knowledge. One of them uses an attribute-value language to represent objects and concepts, and is called *attribute-value learners.* The advantages of attribute-value learners are: relative simplicity, efficiency, and existence of effective techniques for handling noisy data. However, attribute-value learners have two strong limitations:

- the background knowledge can only be expressed in a limited form
- the lack of relations makes the concept description language inappropriate for some domains

Another class of learning systems, called *relational learners*, induce descriptions of relations (definitions of predicates). In relational learners, the languages used to represent examples, background knowledge and concept descriptions are typically subsets of first-order logic. Learners that induce hypotheses in the form of logic programs are called *inductive logic programming* systems.

Compared to attribute-value learners, inductive logic programming (ILP) systems have the following major advantages:

- the use of domain-specific background knowledge enables the user to introduce problem-specific constraints into the learning process
- the representation language is expressive enough to describe the details of many domains

1.3.2 Overview of some ILP systems

Currently, there are three ILP systems which are most widely used. They are: FOIL [23], Golem [25] and Progol [7].

FOIL builds on ideas from both the attribute-value learning algorithms and the use of first-order logic as a representation language. It is the first widely known demonstration that first-order logic can really work on a broad range of problems. FOIL uses efficient methods adapted from attribute-value learners, such as, ID3 [29], and AQ [30]. It can find recursive definitions and develop inexact but useful rules. However, it has the following major limitations: First, FOIL requires all background knowledge to be given extensionally. Second, it is based on a short-sighted greedy algorithm which tends to local generalization.

Golem is a bottom-up algorithm based on Plotkin's notion of relative least general generalization (*rlgg*) [18]. It has been shown to be efficient and effective in many application domains, due to the use of *rlgg* as an underlying hypothesis filter so that it is able to avoid search in the construction of clauses. Golem, however, still suffers from some of the restrictions, such as the prohibition of non-ground unit clauses in the input files, and the restriction to determinate clauses.

Progol is an ILP system which, given examples and background knowledge, constructs hypotheses on the basis of Inverse Entailment [18] and general-to-specific search through a predicate subsumption lattice. Progol is preferred in this study because it allows arbitrary Prolog programs as background knowledge and arbitrary definite clauses as examples.

1.3.3 The Approach

Layout constraints (rules) are used to capture user requirements, and help reduce the search space in the process of layout generation. As mentioned above, our problem is, given a group of good and bad layout examples, to produce generalized layout rules (constraints) from them. Throughout the dissertation, we use layout constraints and layout rules interchangeably.

The specification of the layout rule induction problem can be subdivided into two parts: One is to parse the relationships between different types of YP objects: The other is how to represent the relationships and use them to produce layout rules.

Parsing an arbitrary 2D picture is a challenging task. In our case, the YP layout process is usually constrained by a grid structure which is explicitly or implicitly given by the end user. The grid structure also encodes different priorities for each grid position according to their effectiveness and costs. Hence, we assume all examples are given in a pre-defined grid structure. In this way, parsing an example layout is simplified.

Usually, the nature of an application domain determines the type of learning techniques. The problem of inducing layout rules requires relational background knowledge. In the meantime, layout designers are most interested in the understandability of the induced rules. Therefore, ILP's ability to use background knowledge and representational flexibility makes it the most appropriate technique to be chosen in the domain of layout rule induction. In this work, we decide to choose an ILP system, PROGOL, as the layout rule generator. Input to the Progol system should use first-order logic syntax, so we use Prolog predicates to represent relations between objects.

1.4 System Overviews

In this dissertation, we implement a system that is capable of generalizing layout rules based on a group of good or bad layout examples. The system consists of three components, as shown in Figure 1.3. The parser parses 2D layout examples and generates clauses describing relations between different types of objects. Progol is used to construct layout rules based on the output from the interaction module. The interaction module allows the user to generate input files for Progol based on his requests, and add mode declarations which are necessary for Progol.

Figure 1.4 illustrates a general picture of how ILP can be used in inducing layout

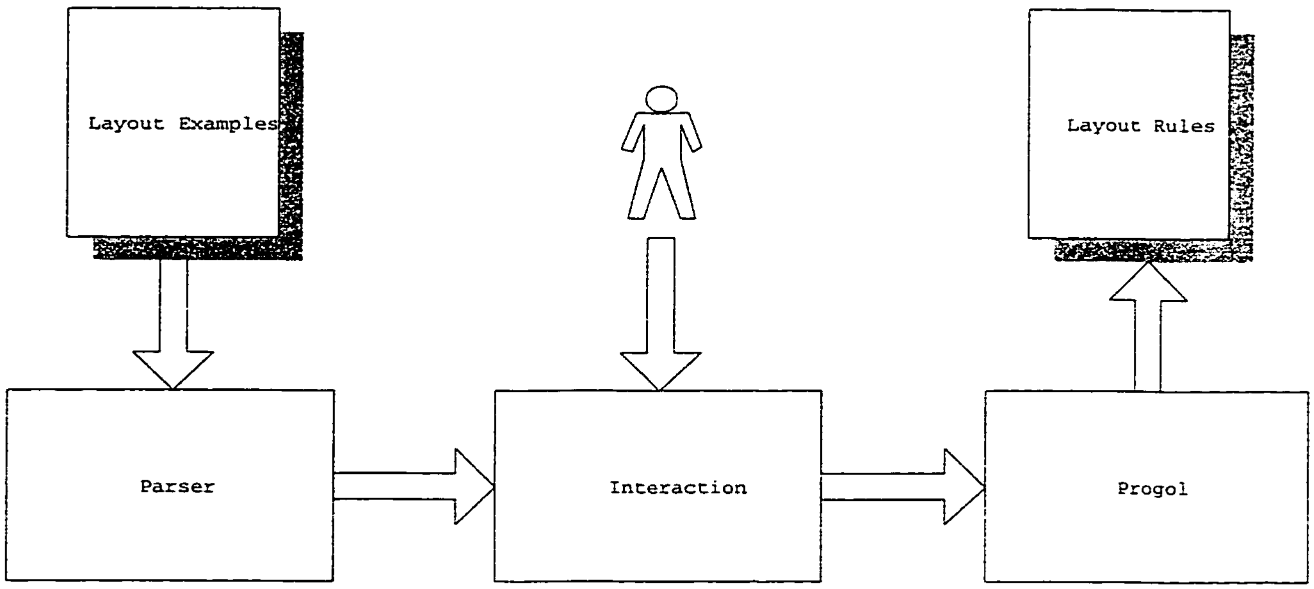


Figure 1.3: **Architectural view of the system**

rules. The ILP system takes as input both relations between YP objects and layout example data and generate possible hypotheses so that those relations together with hypotheses should entail the layout examples.

In Figure 1.4, there are three question marks which point out those different areas which may influence layout rules induction. First, different ILP systems may generate different explanations to the hypotheses and have their own restrictions when applied to layout rules induction. Second, within the same ILP system, different hypothesis restrictions will still affect the rule generation. For instance, suppose we try to induce a layout rule from the layout example shown in the picture, using two different hypothesis restrictions may result in two different rules:

1. layout(A, da, P) $\leftarrow$ aboveICs(A).

2. above(A,B) $\leftarrow$ largeDA(A), smallDA(B).

Rule 1 says that a DA object must be placed above all IC objects, however, rule 2 requires that larger DAs should be above smaller DAs. Those hypothesis restrictions are usually stated using "mode declarations". They enable the user to specify the predicate and function symbols to be considered, and the types and formats of arguments for each. Last, the relation definition language is also important. There are a couple of questions here that we need to consider: The first is how to define a relation so that it can best reflect user's preference and still be understandable? The second is how expressive are the relations so that they can be used to generate layout rules meaningful?

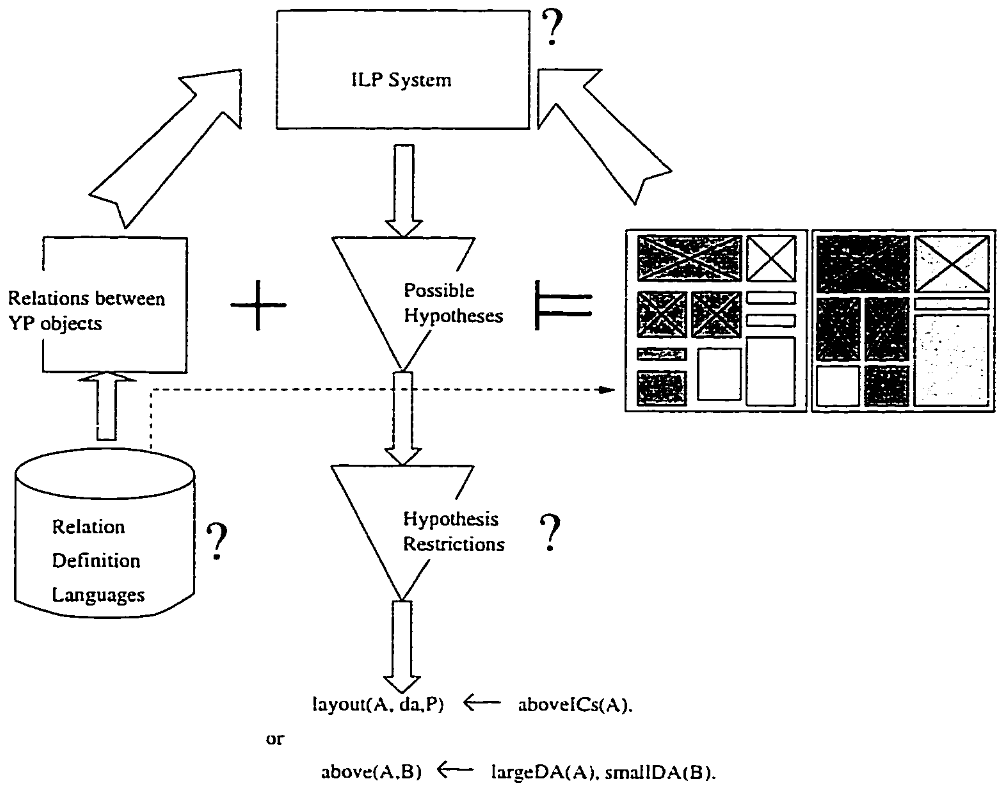

ILP System
?
Relations between YP objects
+
Possible Hypotheses
⊨
Relation Definition Languages
?
Hypothesis Restrictions
?
layout(A, da,P) ⟵ aboveICs(A).
or
above(A,B) ⟵ largeDA(A), smallDA(B).

Figure 1.4: **Functional view of the system**

1.5 The Goal

In order to judge the quality of the induction ability of the system, there are two main directions that we can consider. One is qualitative analysis of the rules, such as, how much insight the rule provides to help the layout designer to understand user's layout preferences, or how easily understandable the rule is. The other is a domain independent measure of the validity of generated rules.

The goal of this work is to investigate the use of one of ILP systems, Progol, to construct layout rules. We want to test if the ILP technique is powerful enough to fulfill the task of inducing layout rules, and whether it can handle the practical layout rule learning problem. Furthermore, we want to accumulate valuable experience of what kinds of relation definitions are necessary in order to represent rules which are expressive enough to state users' layout preferences. Therefore, we focus on the validity check on the first alternative.

From the practical point of view, applying induced layout constraints successfully in YPPS can help automated layout design. The end user does not need to master the declarative YPL language to specify the requirements he prefers, so, it frees layout designers from expressing cumbersome layout constraints. From the theoretical point of view, we make one step forward in determining what is required in order to improve the induction of 2D layout rules.

Chapter 2

Related Work

There are two major research areas which inspire our study of using ILP techniques to generate layout rules. The IMAGE system ([20]) provides a framework to realize application-specific GUIs. It infers a group of mapping rules from multiple examples provided by the programmer. By this way, it saves the complexity and cost of building application-specific GUIs. The other is successful applications of ILP systems in various domains, such as, finite-element mesh design, knowledge discovery in database, and natural language processing.

2.1 IMAGE System

2.1.1 System Overview

In [20], Ken et al proposed an enhanced Programming By Example (PBE) methodology called "Programming by Interactive Correction of Examples", and also implemented a prototype system IMAGE. The new methodology is applied to the generation of mapping rules, which specify the mapping between application data and its corresponding visual representation on GUIs. The study is based on the bi-directional translation model in [21]. In this model, general mapping from abstract application data to pictures, and from pictures to abstract application data, is realized merely by giving declarative mapping rules.

Figure 2.1 shows the system architecture of IMAGE. It consists of four main modules:

Example Generator

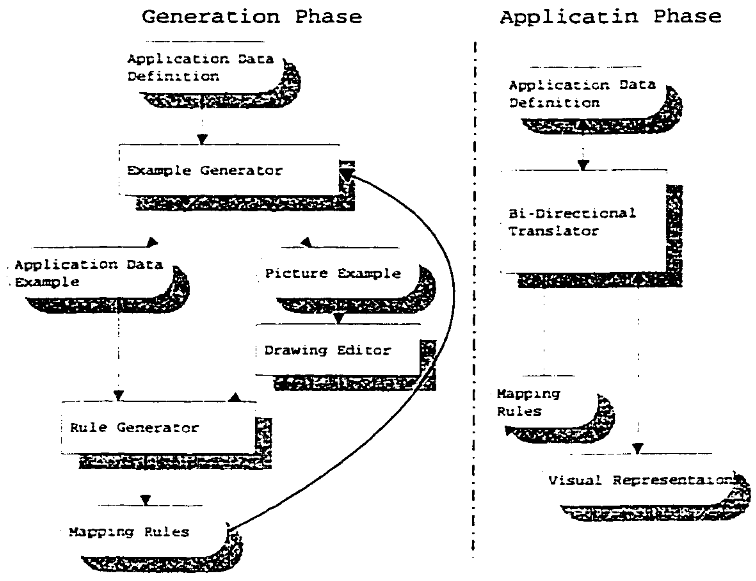


Figure 2.1: **System architecture of IMAGE**

This module is responsible for generating application data examples, given an application data type definition. It usually starts with the simplest application data example, and then revise the examples to make them more complicated by some specified strategies (refer [20]).

Drawing Editor

This module is used as a MacDraw-like editor for the programmers to correct visualized pictures that the system shows.

Rule Generator

Given example data, including application data and its corresponding picture, this module can generalize a mapping rule for translation. Furthermore, if the rule does not reflect the programmer's intention, it can be modified by additional example pairs.

Bi-Directional Translator

This module employs the mapping rules to realize the bi-directional translation between application data and its visual representation, using the hierarchical constraint solver.

IMAGE system has two phases of usage: one is the *rule generation phase*, and the

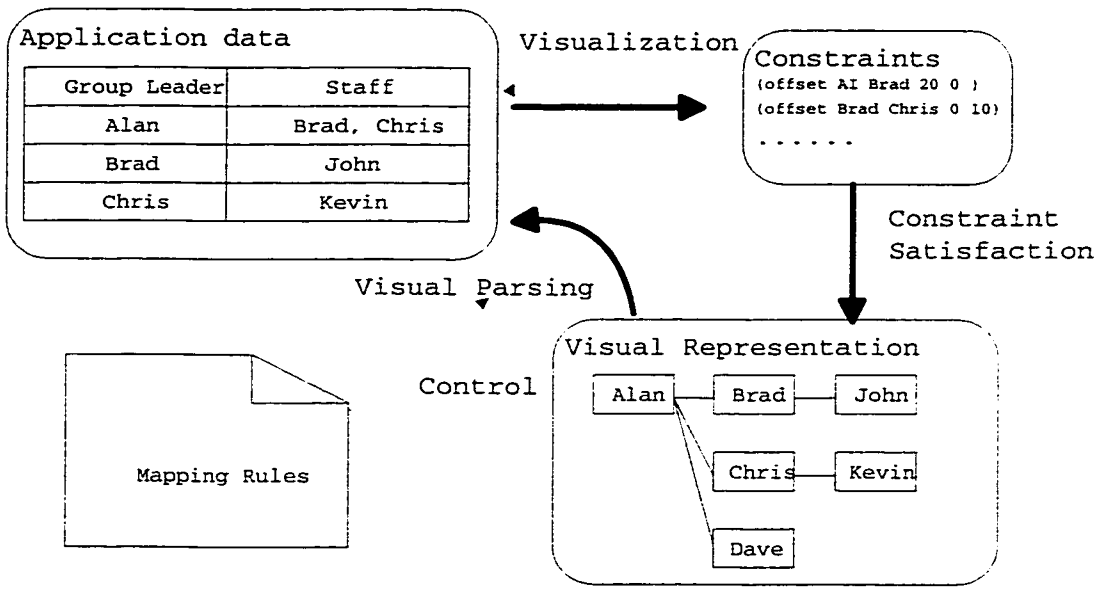


Figure 2.2: **Bi-directional translation model**

other is the *rule application phase*. In the rule generation phase, based on the examples from the example generator and modified pictures from drawing editor, the rule generator produces mapping rules incrementally. When the rule generation process is completed, the application phase is invoked to present the mapping rules visually. In the application phase, the mapping rules control the bi-translation between application data and its visual representation. This phase provide "free-hand" interfaces for end-users to manipulate application data visually.

2.1.2 Bi-Directional Translation Model

Figure 2.2 illustrates the bi-directional model with an example of department diagram data.

Application data consists of a series of structured data. Each data is an instance of a type definition. Visual representation is a picture which is constructed from a series of graphical objects and geometric constraints among those objects. Each graphical element corresponds to application data.

A mapping rule expresses the correspondence between an application data type definition and a series of constraint templates ([20]). As shown in Figure 2.2, mapping rules control the processes of both visualization and visual parsing. When the system visualizes an instance of a data type definition, all the constraint templates in the mapping rule are instantiated, and the constraint solver determines the visual representation by solving the constraints. On the other hand, the visual parsing is also controlled by the mapping rules. It checks whether all constraint templates in the mapping rule can be satisfied in a given picture. If so, the corresponding data type definition is instantiated.

2.1.3 Conclusion

The most important feature is that the system can extract the programmer's intention about the revision of the rules. By applying the rules in visualization, it allows fast and effortless creation of direct manipulation interfaces.Hence, this study motivates our research of inferring layout rules from multiple layout examples provided by the end-user.

2.2 Applications of ILP Systems

ILP systems have been successfully applied to various application domains. In this section, we will present two different application areas of ILP: one is in the area of mechanical engineering (finite element mesh design) [16], another is in the area of knowledge discovery [9]. They both benefit from the ILP's predicate logic descriptions, and from the background facility in ILP. Similarities between the two areas and the problem of YP layout rules induction are: they involve structured data and require background knowledge. Therefore, successful applications in these two domains motivate the use of ILP in inducing layout rules.

2.2.1 Finite Element Mesh Design

The finite element (FE) method is used extensively by engineers and scientists to analyze stresses in physical structures. It requires that the object is partitioned into finite elements, resulting in a finite element mesh. The basic demand for the FE mesh is that the mesh should represent the exact shape of the structure. Since it is very difficult to tell in advance where the mesh should be dense and where it should be coarse because the resolution depends on several factors, including the shape of the structure, the stresses applied to it and the boundary conditions. There exists a strong need for knowledge-based expert systems which are able to help design FE meshes.

FE methods have been applied for the last 30 years. A large number of successful meshes for particular objects have been accumulated. By employing appropriate machine learning techniques, the data from those example meshes can be used to construct a knowledge base for a FE mesh design expert system.

In general, a mesh depends on the geometric properties of the object, the stresses acting on it, and the relations between different components of the object. Also, the mesh density in a region of the object depends on the adjacent regions. All these relational dependencies suggest that the mesh design problem should apply ILP techniques.

In [16], the ILP system GOLEM is employed to induce rules for deciding on appropriate resolution values of FE meshes. The resolution of a FE mesh is determined by the number of elements on each of its edges. Thus, the problem of learning rules for determining the resolution of a FE mesh can be formulated as a problem of learning rules to determine the number of elements on an edge.

The input data for GOLEM consisted of 75 foreground examples, 618 negative examples, and 588 background examples. The results obtained with GOLEM were satisfactory even though some of the rules were useless. Here, we give an example of a rule (in Prolog syntax).

```
mesh(Edge, 7) :-
    usual_length(Edge),
    neighbor_xy(Edge, EdgeY),
    two_side_fixed(EdgeY),
    neighbor_zx(EdgeZ,Edge),
    not_loaded(EdgeZ .
```

This rule says that partitioning *Edge* into 7 elements is appropriate if *Edge* has "usual length", and has a neighbor *EdgeY* in the xy-plane so that *EdgeY* is fixed at both ends, and *Edge* has another neighbor *EdgeZ* in the xz-plane so that *EdgeZ* is not loaded.

2.2.2 Knowledge Discovery in Databases

In [9], a framework for realizing knowledge discovery in database by utilizing ILP technology is proposed. Most of the researches on knowledge discovery in databases are based on decision tree algorithms. The decision tree approach is efficient in extracting rules, however, its propositional logic representation formalisms make it difficult to take into account the available background knowledge. In the domain of knowledge discovery in databases, this limitation makes a big difference. Namely, adequately prepared background knowledge enables us to generalize examples in a more natural and concise manner. On the other hand, ILP approach can handle background knowledge very naturally since it is based on first order logic.

Given a database and an inductive inference problem, how to design a target concept representation is important. In the database terminology, other issues need to be considered, such as, how to identify necessary background knowledge, how to generate negative examples automatically. Shimazu [9] solved the issues by proposing an extended Entity Relationship model, called RER model, to define the database. By translating the database into the RER model, relevant database data can be converted to input data for Progol. In addition, background knowledge is made from each entity of RER model and relationship. The system employed the ILP system, Progol, to induce rules.

Moreover, [9] presented an experimental study of automatic knowledge acquisition of expertise in an e-mail classification expert systems. The results showed that the system was able to extract a set of useful rules for 18 classes out of 20. Those rules can be used as knowledge bases in an e-mail classification expert system.

Chapter 3

Use of Progol to Construct Layout Rules

In this chapter, we use an example to demonstrate how Progol works with given examples. As the goal of our work is to use Progol to generalize layout rules from a sequence of layout examples, we will give a detailed discussion of how to represent layout rules using Prolog clauses.

3.1 Data used by Progol

Progol is an ILP system that mixes "learning by being told" and "learning by examples" because it can make use of *background knowledge* that is supplied by the teacher and expressed in Prolog clauses. Input to Progol consists of mode declarations, background knowledge, and positive and negative examples. In this section, we will use a simple example to explain how Progol works with these three types of input. The example here comes from [8] in that further details about Progol are discussed.

In the example, we are given a sequence of trains (see Figure 3.1). Each train has attached a number of cars, each of which may have a number of different properties such as being long or short, having a roof or no roof, and having a shape painted on the side. In addition, we are given that each train is either traveling east or traveling west. The problem is to find a rule that will predict, from the properties of its cars, which direction the train is traveling.

A file containing the data for the trains example is given in Figure 3.2

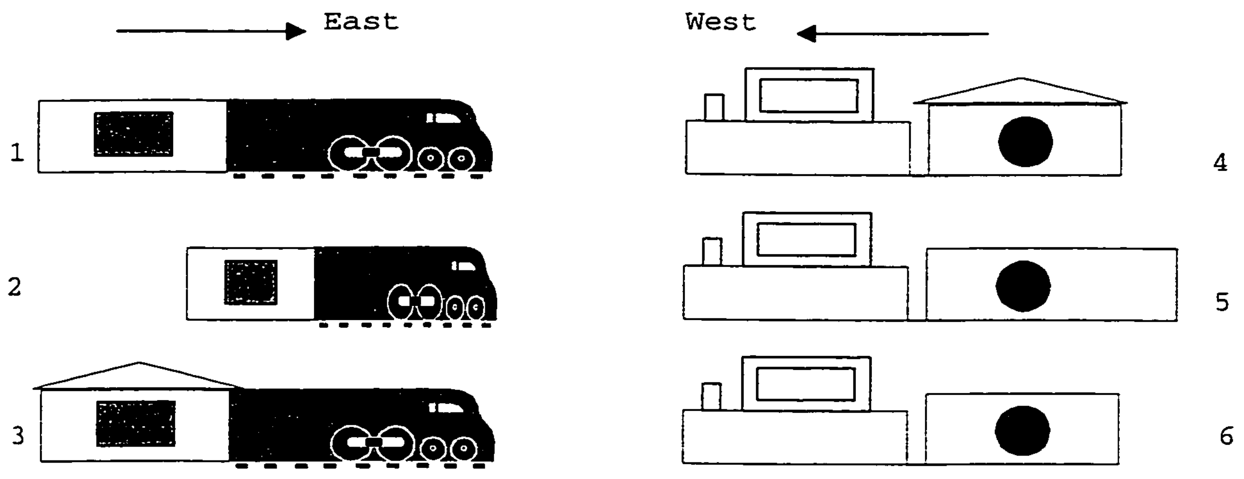


Figure 3.1: **Eastbound and westbound trains**

```
%% Mode Declarations

:- modeh(1, eastbound(+train))?
:- modeb(1, nextcar(+train,-car))?
:- modeb(1, shape(+car, #shape))?

%% Types

train(train1).  train(train2).   train(train3).
train(train4).  train(train5).   train(train6).

car(car1_1).  car(car2_1).  car(car3_1).
car(car4_1).  car(car5_1).  car(car6_1).

shape(rectangle).   shape(circle).

%% Background Knowledge

nextcar(train1, car1_1).   nextcar(train2, car2_1).
nextcar(train3, car3_1).   nextcar(train4, car4_1).
nextcar(train5, car5_1).   nextcar(train6, car6_1).

shape(car1_1, rectangle).  shape(car2_1, rectangle).
shape(car3_1, rectangle).  shape(car4_1, circle).
shape(car5_1, circle).     shape(car6_1, circle).

%% Positive Examples
eastbound(train1).
eastbound(train2).
eastbound(train3).

%% Negative Examples

:- eastbound(train4).
:- eastbound(train5).
:- eastbound(train6).
```

Figure 3.2: **The file `trains.pl`**

First,

```
:- modeh(1, eastbound(+train))?
```

As we have discussed, Progol constructs general rules expressed as Prolog clauses, which have a *head* and *body*. This mode declaration says that general rules may have heads containing `eastbound(X)`, where `X` is a variable of type `train`. The number `1` is the *recall* of the mode declarations. It is used as a bound on the number of alternative *instantiations* of the predicate that Progol will consider. An instantiation of the predicate is a replacement of the types by either variables or constants in accordance with the `+`, `-`, and `#` information. If we know that there are only a certain number of solutions for a particular instantiation, we can express this as a recall parameter to save Progol searching fruitlessly for further solutions.

Secondly,

```
:- modeb(1, nextcar(+train,-car))?
:- modeb(1, shape(+car, #shape))?
```

are the mode declarations describing the form of literals that can be used in the body of a Progol hypothesized clause. The first says that the general rules may have bodies containing the predicate `nextcar(X,Y)`, where `X` is of type `train` and `Y` is of type `car`.

The + types are used where there is an input argument of a predicate, and - types are used for an output argument. In cases where arguments can be both input and output, two mode declarations can be given.

In the second declaration, the # in `#shape` type says that we must have *constants*, not variables, of type `shape`. Thus we can have rules with bodies containing `shape(B,rectangle)` but not `shape(B,C)`.

Thirdly, in the Progol input file, we must add some additional types and background information in order to do generalization.

For instance,

```
shape(rectangle).  shape(circle).
```

is a statement to specify the extension of `shape`. In this case, it says that the type `shape` can only have two values, one is `rectangle`, the other is `circle`.

```
  nextcar(train1, car1_1).
```

```
shape(car1_1, rectangle).
```

The above two statements give us necessary background facts about train1. That is, train1 has one car, car1_1, and the car has a shape of rectangle on its side.

Finally, a positive example is a statement such as:

```
eastbound(train1).
```

A negative example is a statement such as:

```
:- eastbound(train4).
```

The fact that the example is negative rather than positive is marked by the occurrence of :- before the example.

With this information relating to the trains, we can ask Progol to generalize from these examples and produce a more general rule. What Progol does can be divided into three parts. The first part is to construct the *most specific clause* of the first positive example. In this case, the first positive example is eastbound(train1), and Progol will get its most specific clause using mode declarations:

```
eastbound(A) :- nextcar(A,B), shape(B,rectangle).
```

The second part is to make new clauses by combining the predicates contained in the most specific clause. When Progol applies best first search and finds a clause which explains most of the positive examples (and none of the negative ones), it selects this clause as its general rule. In this case, the general rule is the most specific clause itself.

The third part is to add this general rule to its information concerning the trains and remove any of the original positive examples which are now redundant. In the example, the general rule explains all the positive examples and all three are removed. Progol finishes by printing out the rule concerning the eastbound predicate. That is,

```
eastbound(A) :- nextcar(A,B), shape(B, rectangle).
```

It tells us that a train is heading east if its first car has a rectangle on it.

3.2 A study of layout rules

The goal of our system is to use the ILP system, Progol, to generalize layout rules from a group of given layout examples. In order to do that, we need to parse the known

Term	*Explanation*	*Abbreviation*
IC	Incolumn object	IC
Heading	Heading object	HD
IF	Incolumn filler object	IF
DF	Display filler object	DF
HPDA	Half page display ad	HPDA
FPDA	Full Page display ad	FPDA
small DA	Display ad object not including HPDA and FPDA	DA

Table 3.1: **Explanations of terms**

layout examples and explore implicit relationships between different types of objects. There are a couple of questions that are important to us. How do we represent a layout rule so that it can best reflect the user's intention? How many relations are necessary in order to generate useful layout rules?

We begin our explanation by studying layout rules from a real YP publishing company. Before we start, we need to explain some terms that are used in YP publishing.

3.2.1 Terminology

Throughout the rest of this dissertation, certain terms are used with enough frequency to warrant a clear understanding of their meaning with respect to the YP automated layout problem. This section is intended to provide a common ground for the intended meaning of these terms.

1. YP Objects

   Table 3.1 explains terms that we use to represent different types of YP objects.

2. a grid structure

   It is known that a general layout problem is strongly *NP-complete* and thus, there is no general and efficient algorithm for solving it([5]).The problem becomes linear in time only if we use additional constraints to reduce the design space for arrangements, e.g., a grid structure.

   A grid partitions a 2D plane into smaller rectangular units of mostly equal size using horizontal and vertical lines. Figure 3.3 shows a weighted grid which is used to control pagination process in YPL. A page contains 8 rows and 4 columns. Each grid has a weight which shows its priority according to its effectiveness and costs. The lower value of a grid field, the higher priority it represents. Throughout the dissertation, we assume all layout example objects are placed within the 8-row and 4-column grid structure.

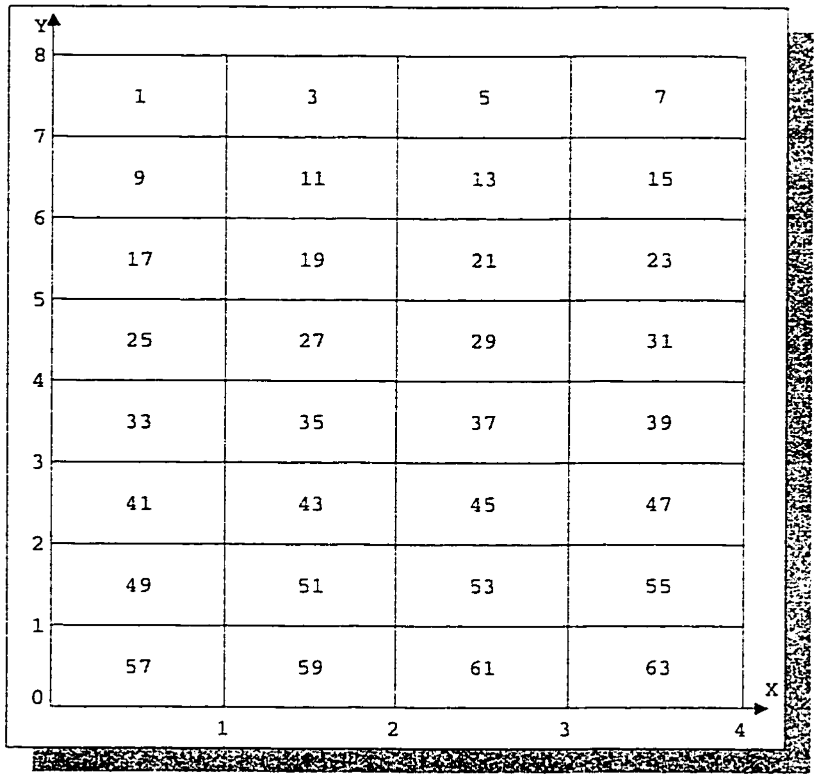


Figure 3.3: The weighted grid structure

3. rank

   We use "rank" to order each DA object based on its size. For example, if we use the grid shown in Figure 3.3, then the size of an HPDA is 16. Thus, its rank is 16.

4. scope

   The concept of scope is critical as various scopes defined for YPPS can expedite the design of layout constraints and their efficient application. In the dissertation, scope means either page or spread, depending on the current layout scope.

3.2.2 Layout Rules in the Real World

A directory publishing company typically uses layout rules to express its layout requirements. Such layout rules can be translated to corresponding layout constraints which can be applied to a stream of display objects in order to achieve aesthetically appealing and rationally organized layouts. As pointed out in [4], it is very important to mention the distinction between the notion of a valid layout and a quality layout in the process of applying constraint technology to the design and construction of a feasible automated YP layout,

A validity constraint is typically applicable in a straightforward way, such as.

(1) There must be no IC above a half page display ad.

With the technique of constraint propagation, validity constraints can be used in reduction of the solution space.

However, another such informal rule like :

(2) A DA must be placed as close as possible to its heading.

introduces the issue of "...as close as possible...". We call such a constraint a "quality constraint". It is typically used as an optimization heuristic in order to distinguish more preferable layouts.

Our aim is to study user-defined layout examples and extract useful expressions of the relationships between different types of objects. For our study, we concentrate on layout rules which address validity criteria through some examples. Therefore, we limit our study to validity constraints and leave quality constraints as further work.

Here are some real layout rules from a directory publishing company.

1. SmallDAsAndICsStartOnSamePage

   Consider the first page where ads of heading *h* (e.g. cycle shops) are placed. If this page contains a small DA, then there must also be placed some ICs of that heading.

   Figure 3.4 shows a bad example of this rule. There are no ICs of heading h2 on page 141 although page 141 contains three DA objects of heading h2.

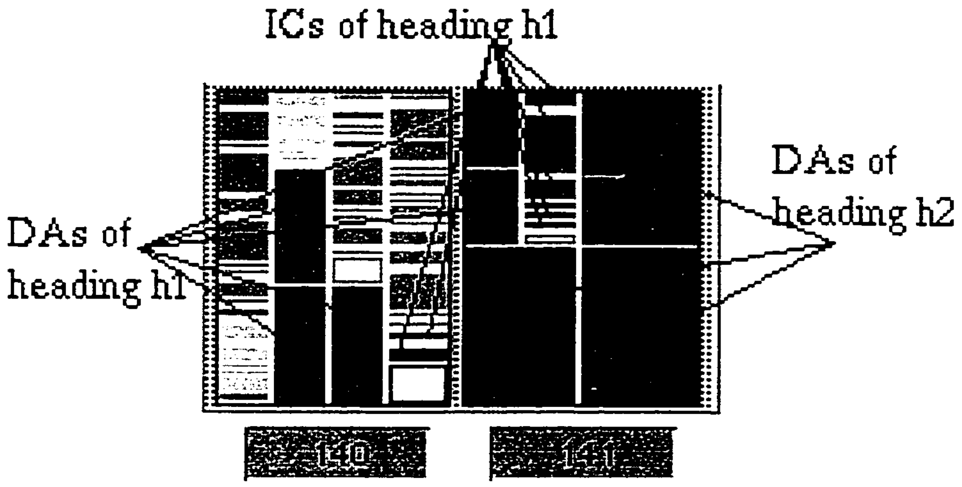


Figure 3.4: **A bad example of rule: SmallDAsAndICsStartOnSamePage**

2. NoLonelyDA

   If a DA is smaller than a half page and if it is not the only DA of its heading, it must not be placed as the only DA of its heading on a page. In other words, if a heading has more than one small DA, on each page there must be at least two of them. For example, in Figure 3.5, page 123 has two DA objects a1 and a2 which are of the same heading *h* and page 124 has six DA objects of the same heading *h*. Since there are more than one DA of heading *h*, there must be more than one DA object on each page. A bad example of this rule is provided in Figure 3.6.

3. NoIcUnderDA

   There must be no ICs under DAs. It requires that all DA objects must be placed under IC objects.

4. NoICsAboveHPDA

   Above a Half Page DA (HPDA) there must be no IC. This rule could also be called "HPDAsOnTopIfPossible" because from the viewpoint of the implementation it is enough to force a HPDA on top of the page if there is no other HPDA. A bad example of this rule is given in Figure 3.7. The HPDA object on

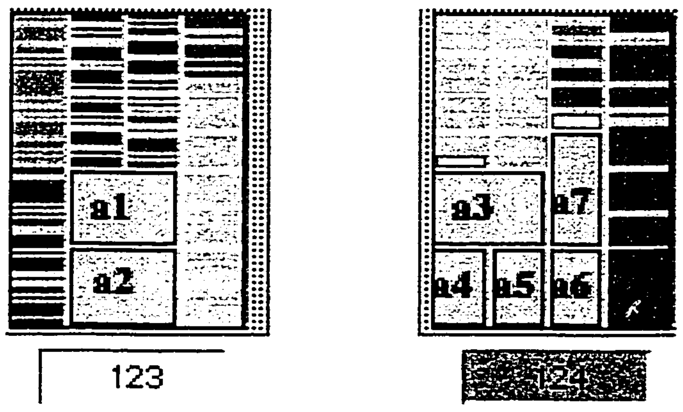


Figure 3.5: A good example of rule: NoLonelyDA

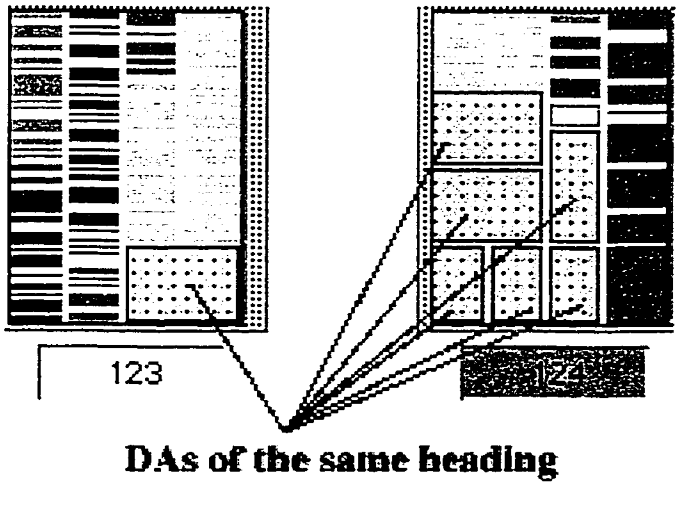


Figure 3.6: A bad example of rule: NoLonelyDA

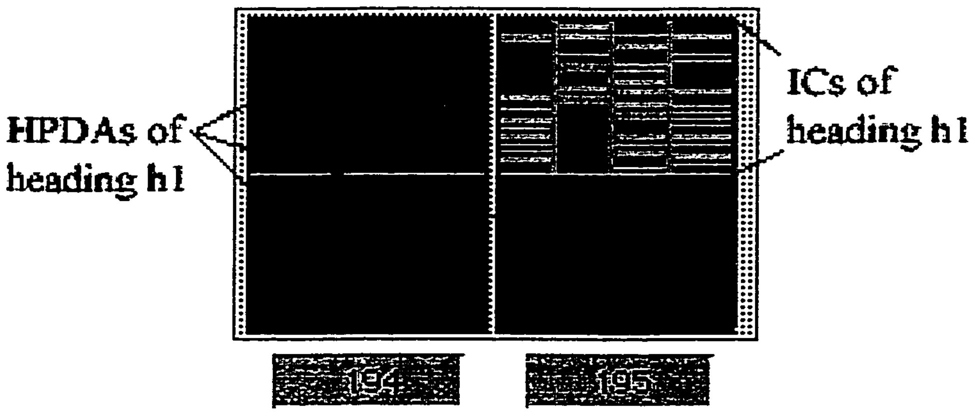


Figure 3.7: **A bad example of rule: NoICsAboveHPDA**

page 195 is placed wrong because it is under IC objects.

5. NoICsBeforeHalfPageOrFullPageDA
   When a heading contains a DA that has the size of a half page (HPDA) or a full page (FPDA), these ads must be the first placed objects of this heading. That means only the heading itself together with other filler objects are placed before those objects. As shown in Figure 3.8, objects which are allowed to place before HPDAs are either heading object or filler objects.

6. NoIFsAtTopOfPage
   Incolumn filler (IF) must be placed at the top of a column. A column must either start with a normal IC or with DAs of DA-like objects.

7. DFsOnlyAtBottomOrUnderDA
   Display filler must only be placed under a DA or on positions that touch the bottom.

8. Touching
   Each DA of a heading must transitively touch an IC of its heading. In other words, a DA touches, if and only if it touches an IC of its heading or another

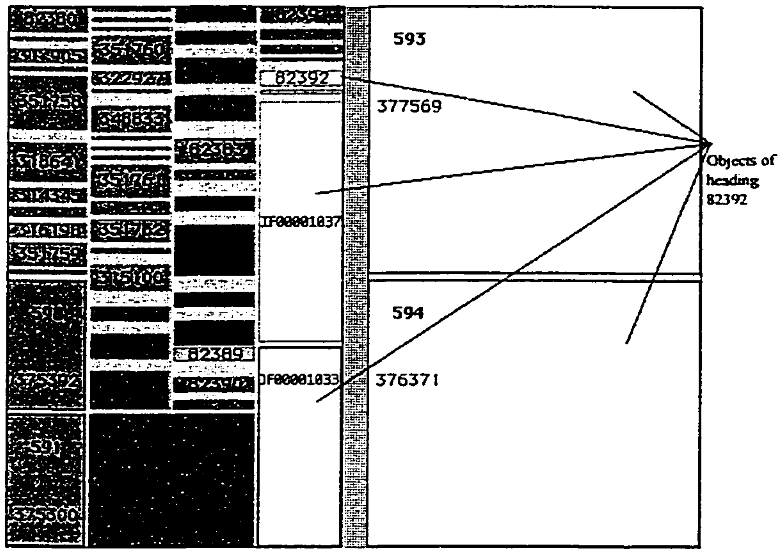


Figure 3.8: **A good example of rule: NoICsBeforeHalfPageOrFullPageDA**

DA which touches that heading. We consider incolumn filler (IF) under an IC of heading Hg to belong to Hg. Note that the "scope" of touching is always a page, not a spread and touching over display filler (DF) is not allowed. The length of the line at which two DAs must touch each other when they touch on their sides is controlled by a parameter. Figure 3.9 shows a bad example with regard to the rule of "touching". Note that objects that are of the same heading are filled with the same color and there are only two DA objects in Figure 3.9: object 378424 and object 378684. The first DA belongs to heading 82328 and it does not touch any object of that heading.

9. RankByGridWithinHeading

   When there are two DAs of the same heading on the same scope, then the DA with the better rank must be on a better position than the DA with the worse rank. The quality of a position P is reflected by the grid weight of the grid field in the upper-outer corner of the DA at position P. Figure 3.10 is a bad example of this rule. The reason why it violates the rule is that the DA object 376184 and another DA 378289 are placed wrong. Since object 378289 has a better rank than the object 376184, it should take a position which has a higher priority. However, the object 378289 has a priority of 45 and object 376184 has a priority of 43, which is better than another DA.

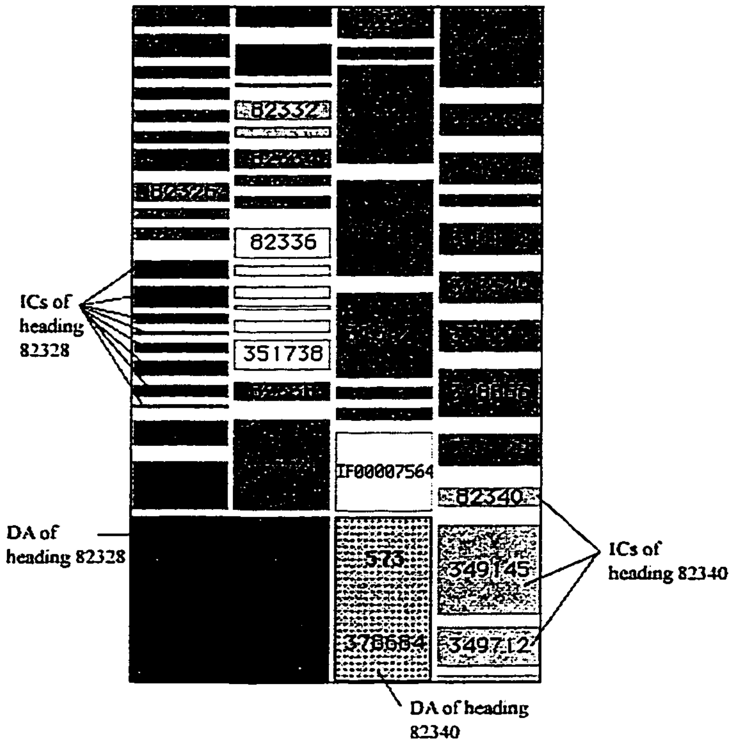


Figure 3.9: A bad example of rule: Touching

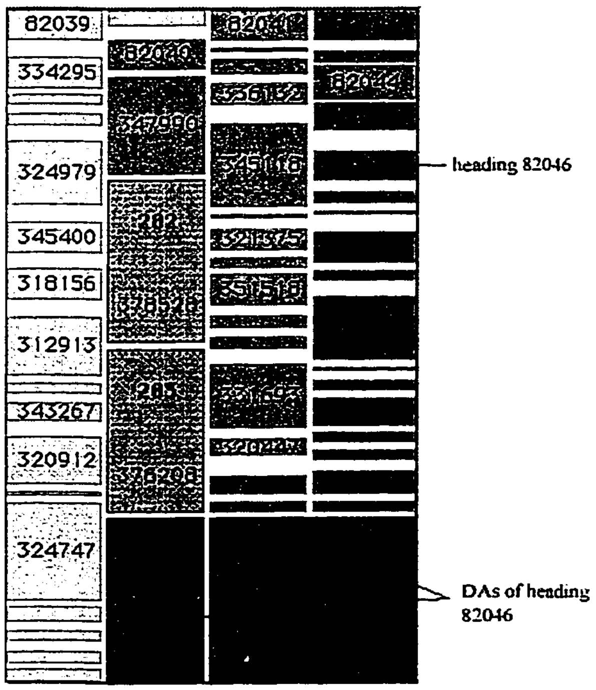


Figure 3.10: A bad example of rule: RankByGridWithinHeading

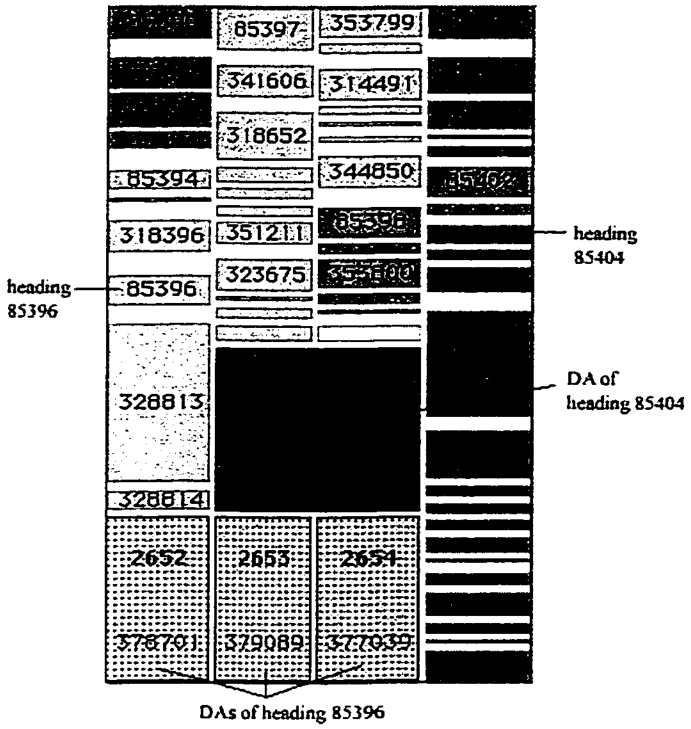


Figure 3.11: **A bad example of rule: ReadingOrder**

10. ReadingOrder

    If there are two DAs of two different headings on a scope, then the DA of the earlier heading must be placed before the DA of the later heading in reading direction. Figure 3.11 shows a bad example concerning this rule. There are four DA objects in Figure 3.11. Object 2652, 2653, 2654 are of the same heading of 853964 whereas object 2656 belongs to another heading 85404. According to the rule, object 2656 should not be placed above object 2653, 2654.

3.2.3 Representing Layout Rules

As we know, Progol uses Horn clauses to represent the results of learning. In this section, we discuss the use of Horn clauses to represent the individual rule given above.

Object Type	Category
DA	DADFs
DF	
IC	ICIFs
IF	
HD	hd
HPDA	hpda

Table 3.2: **Classification of categories**

Relations

In order to represent the layout rules, we need to determine what relations are required from the layout examples, and how to express them. The need for a relation is determined essentially by its interestingness to a layout designer and its clarity in helping to express those rules we discussed before.

Here, we introduce a new concept "category". Table 3.2 describes the classification of categories. In this case, we consider geometric inter-relationships between two object categories rather than its types. For example, the relation "underICIFs(A, P)" is more interesting than a relation of "under(A, B)". The relation "underICIFs(A, P)" means that an object A must be placed under objects of category "ICIFs" on a page P. Usually, a layout designer pays more attention to the geometric relationship (i.e. "under" or "above") between two object categories rather than two individual objects.

Here follows a detailed description of relations we use in our study:

1. icNum(H, N, P)
   On page P, the number of ICs of heading H is N.

2. daNum(H, M, P)
   On page P, the number of DAs of heading H is M.

3. icdaSamePage(icNum(H,N1,P),daNum(H,N2,P)).
   When a number of ICs and small DAs of the same heading are on the same page, we can use the above relation to represent that it is consistent with the rule of "SmallDAsAndICsStartOnSamePage".

4. das(List, H)

   In order to represent the rule of "NoLonelyDA", we use a list *List* to record the set of DAs of a specific heading H on each page. For example, relation `das([2,3], h80252).` means that five DAs of heading h80252 are placed on two separate pages. The first page contains two DAs and the second has three DAs. It conforms to the rule of "NoLonelyDA" because each page has more than one DAs of heading h80252.

5. nolonelyDa(das(L, H))

   We use "das(L,H)" to record the distributions of DAs of heading H on each page. If the placement of DAs conforms to the rule of "NoLonelyDA", then we have this relation.

6. underICIFs(A,P)

   On page P, an object A is always placed under objects whose type are either IC or IF. This relation is useful when expressing the rule of "NoIcUnderDa".

7. aboveICIFs(A,P)

   On page P, an object A is always placed above objects whose type are either IC or IF.

8. aboveDADFs(A,P)

   On a page, an object A is always above objects which are either DAs or DFs.

9. underDADFs(A,P)

   An object A is always under objects which are either DAs or DFs.

10. abovehpda(A,P)

    An object A is above an HPDA object on page P.

11. underhpda(A,P)

    An object A is under an HPDA object on page P.

12. downNext(A, T)

    T should be a specific type of those defined types, i.e., da, ic, hpda, if, df, or hd. It says that an object A is under another object which is of type T and A is also adjacent to that object.

This relation considers a relationship between one object and another object type. We need to define a relation that a DF object must be under and adjacent to a DA object because the rule cares only another object's type. Therefore, we abandon another alternative form like "downNext(A,B)", in which A and B are both objects.

13. orderBefore(A,hpda,H)

    In order to represent rule 5 "NoICsBeforeHalfPageOrFullPageDA", we need to analyze which objects are before an HPDA object. We need only consider those objects which are of the same heading as the HPDA object. This relation is used to specify that object A is on a previous page than the HPDA object. H is the heading of both the HPDA object and the object A.

14. orderBefore(A,fpda,H)

    A is the object which can be placed before a FPDA object of its heading.

15. topCol(A, P)

    Since all objects are placed within the grid structure, each object must belong to one of the four columns. An object could be placed in one of the three positions with regard to a column: top, middle or bottom of a column. If it is at the top of a column on page P, then the relation of "topCol(A,P)" holds for that object.

16. midCol(A, P)

    If the object A is placed in the middle of a column, it does not touch the top nor touch the bottom, then the relation "midCol(A,P)" holds.

17. touchBot(A, P)

    If the object A is at the end of a column, which means, it touches the bottom of a column on page P, then object A has the relation "touchBot(A,P)".

18. touch(A,B)

    Here, A and B represents two different objects. The relation of "touch" has two requirements. First, it means that object A and object B share edges. Then it requires that object A and object B are of the same heading. For example, in Figure 3.12, object *a* touches *c* because object *c*'s upper line touches object *a*. Note we also consider two objects have a "touch" relation if they share an endpoint. As shown in Figure 3.12, objects *b1*, *b2*, *b3* and *b4* all has only one endpoint which touches object *a*. In this case, we think that there exist "touch" relations between *b1*, *b2*, *b3*, *b4* and *a*.

It is also important to point out that relation "touch" is transitive. If an object A touches B, and B touches another object C, then we have the relation of "touch(A,C)".

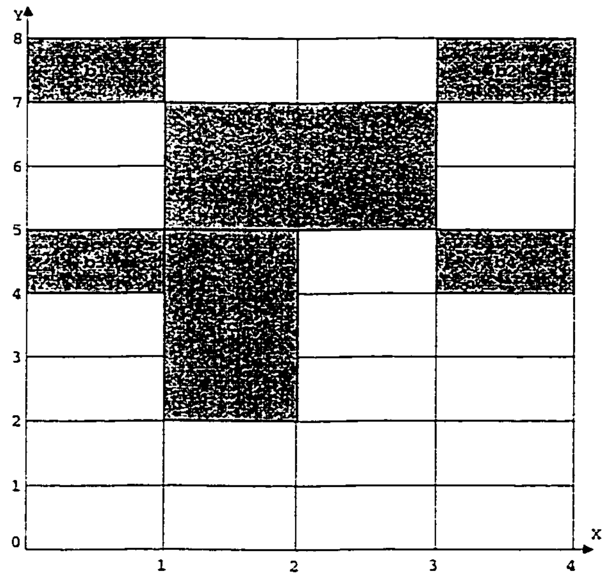


Figure 3.12: **An example of "touch" relation**

19. betterPosition(A, B)
    Object A is in a better position than B. In the pagination business, a position which a DA takes often reflects its cost and effectiveness. Usually, the largest DA should be placed on the best position. Therefore, when considering which object takes a better position than another, we restrict the scope to DA objects.

20. rank(A, N)
    We use "r(A, N)" to represent that a DA object A has a rank of N. We use the size of the DA object as its rank.

21. piece(A, T, H, P)

    Object A belongs to type T. It is of heading H and on page P.

22. readingBefore(A, B)

    In a reading direction, object A is placed before object B. Here, we consider that the reading order is from left to right and from top to bottom.

23. layout(A, T, P)

    This relation means that an object A of type T is in a valid position on page P. Usually, type T is a specific type, such as, IC, DA. We use this relation to represent an attribute of object A. For example, "layout(A,da,P)" tells us that object A is an DA object and on page number P. Meantime, object A is in a valid position which does not violate the layout rule of how to place a DA object.

24. goodTouching(piece(A, T, H, P))

    In YP publishing, it is often required that an object of one type touches an object of another type. If not, then we think that object does not comply with the rule of "Touching".

    We use the above relation to represent that an object A is in a good touching position. In other words, it tells us that object A is placed in a position which complies with the rule of "Touching". For a good layout example of "touching", we assume that each object in the example has the relation "goodTouching".

All relations above except relations $3, 5, 23, 24$ could be parsed from provided layout examples. The relations $3, 5, 23, 24$ can only be determined from good examples with regard to a particular rule. For instance, if the user gives a good example of the rule "touching", then we can assume each DA object in the layout has the relation "goodTouching". On the other hand, in a bad example of "touching", we can only have the relation ":-goodTouching" for each DA object.

Moreover, we divide the above relations into two categories: one is so called "attribute relation", another is "relative relation". *An attribute relation* is a relation which does not relate to a specific object. It gives us the information about the number of objects of different types on the page or other information about a heading. For

Attribute Relations	Relative Relations	
	class one	class two
icNum daNum das	topCol midCol touchBot underICIFs abovehpda aboveDADFs underDADFs aboveICIFs underhpda	downNext touch orderBefore readingBefore betterPosition

Table 3.3: **Classification of relations**

example, the relation "icNum(H,N,P)" tells us how many IC objects of heading H on page P.

A relative relation talks about the relationship of one object with another object in the aspects of positioning, reading order, touching, etc. Furthermore, some of them can represent objects' relative positions on a page, such as, "topCol" and "touchBot". Relative relations can be further divided into two classes: A relation in class one tells us a relationship between one object and the page. A relation in class two is used to describe a relationship between two objects.

Table 3.3 describes the classification of the relations.

Representing the rules

Given the above relations, we can represent the layout rules in section 2.1 using Horn clauses.

1. Rule 1: SmallDAsAndICsStartOnSamePage

   ```
   icdaSamePage(ics(H, N, P), das(H, M, P)) :- N > M.
   ```

   "icdaSamePage(ics(H, N, P), das(H, M, P))" is the *head* of the clause and "$N > M$" is the *body* of the clause. The rule tells us that if small DAs and ICs of one heading are placed on one page, then the number of ICs on that page must be greater than of small DAs.

2. Rule 2: NoLonelyDA

   ```
   nolonelyDa(das([A], H)).
   ```

```
nolonelyDa(das([A|B], H)) :- A>1, gt(B,1).
```

The definition of `gt([A|B], H)` is:

```
gt([ ], _).
gt([H|T], A) :- H > A, gt(T, A).
```

The above expression tells us two conditions which conform to the rule of "NoLonelyDA":

- all DAs of a heading are placed on one page
- If a heading H has more than one DA, then the final layout must have at least more than one DA on each page.

3. Rule 3: NoIcUnderDA
   ```
   layout(A,da,P) :- underICIFs(A,P).
   ```
   On a page, a DA object must be placed under all IC or IF objects.

4. Rule 4: NoICsAboveHPDA
   ```
   layout(A, hpda, P) :- aboveDADFs(A,P).
   layout(A, hpda, P) :- abovehpda(A,P).
   layout(A, hpda, P) :- downNext(A, hpda).
   ```
   HPDA is a special case of DA object, therefore, it must comply with Rule 3. Combining Rule 4 with Rule 3, we encode the restriction that only when an HPDA object is above DA(DF) objects or above another HPDA object, should it be in a valid position. There exists the third case that an HPDA is still in a valid position, which is, the HPDA object is under another HPDA object.

5. Rule 5: NoICsBeforeHalfPageOrFullPageDA
   ```
   orderBefore(A, hpda, H) :- piece(A, df, H, P).
   orderBefore(A, hpda, H) :- piece(A, if, H, P).
   orderBefore(A, hpda, H) :- piece(A, hd, H, P).
   orderBefore(A, fpda, H) :- piece(A, df, H, P).
   orderBefore(A, fpda, H) :- piece(A, if, H, P).
   ```

`orderBefore(A, fpda, H) :- piece(A, hd, H, P).`

The above rules show that only heading and filler objects of the same heading as the H(F)PDA object can be placed before it.

6. Rule 6: NoIFsAtTopOfPage

   `layout(A, if, P) :- midCol(A, P).`

   `layout(A, if, P) :- touchBot(A, P).`

   Since an IF object cannot be placed at top of a column, it can only be either in the middle of a column or touch the bottom of a column.

7. Rule 7: DFsOnlyAtBottomOrUnderDA

   `layout(A, df, P) :- downNext(A, da).`

   `layout(A, df, P) :- touchBot(A, P).`

   Display filler must only be placed under a DA or on positions that touch the bottom.

8. Rule 8: Touching

   `goodTouching(piece(A, da, H, P)) :- touch(A,B), piece(B, ic, H, P).`

   `goodTouching(piece(A, da, H, P)) :- touch(A,B), piece(B, if, H, P).`

   Only when a DA object touches an IC(F) object of its heading, should we consider it to be in a good "touching" position.

9. Rule 9: RankByGridWithinHeading

   `betterPosition(A,B) :- rank(A,C), rank(B,D), sameHeading(A,B), D=<C.`

   If object A's rank is greater than that of object B, and also if A and B are of the same heading, then it must be placed at a better position.

10. Rule 10: ReadingOrder

    `readingBefore(A,B) :- headBefore(A,B).`

    Here, we introduce a new relation "`headBefore(A,B)`". Its definition is:

    `headBefore(A,B) :- piece(A,_,H1,_), piece(B,_,H2,_), H1<H2.`

    That means object A's heading is less than object B's heading in alphabetical order. This rule compares two DAs of two different headings on the same page.

Since these relations are sufficient to represent each layout rule, our system must be able to find these relations in examples. Then we can test if the ILP system, Progol, is able to generalize the rules that we expect. From the experimental point of view, it is an indispensable step. Furthermore, we may change the mode declarations in a Progol file to investigate whether Progol can construct some new and interesting rules.

Chapter 4

Design of the Implementation

This chapter is concerned with the details of the design of the implementation of the system. First, we discuss the use of a flat file format to represent a two-dimension layout example. After we give a brief introduction to the system, we discuss the design of the parser. In addition, we describe a method for generating an input file for Progol in the interaction module. Finally, we will discuss some implementation issues.

4.1 A working assumption

Figure 3.3 shows a grid structure that is used to control pagination process in YP layout. We assume that all 2D layout examples are given based on the grid structure. One grid corresponds to one page. Figure 4.1 shows an example of a spread. Each YP object is represented by a box, and located by its upper-left corner. For instance, the heading object labeled by "82011" is located at position (0, 8); the DF object labeled by "df821" is at position (3, 2). The shading is used to distinguish objects of different headings. Objects belonging to the same heading are filled with the same shading.

We use flat files to represent different layouts of YP objects. Each file has a fixed format. The layout in Figure 4.1 can be translated into a file as shown in Table 4.1. In the beginning of the file, there is one number that specifies how many pages we represented in the file. To represent a layout on a page, the file begins with the page number and the number of columns on the page. Then it lists how many objects for

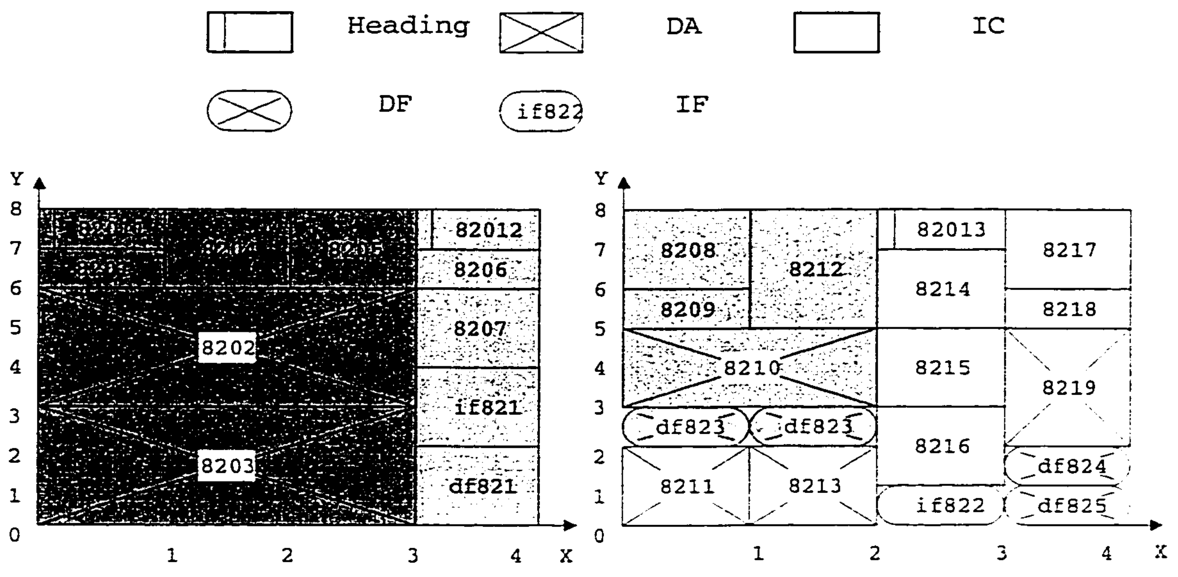


Figure 4.1: **A layout example**

each column, and detailed information of each object in the current column. Each line represents a YP object. It contains: objectID, type, which heading it belongs to, width, height, x and y coordinates. Each attribute is separated by a space. Note that we classify different objects into different columns by their x-coordinate. If an object A is located at (n, m), then we assume that it belongs to column n. Since the flat file can represent the layout and describe attributes of each YP object in the layout example, we can use it as input, and compute the relations between different YP objects.

4.2 Overview

In Chapter 1, we mentioned that the system has three components: parser, interaction module, and the Progol system. The integration is illustrated in Figure 4.2.

The primary purpose of the parser is to analyze whether the relations discussed in Chapter 3 are satisfied in the current layout. If so, the output of the parser should include a set of analyzed relations. The input is a file which expresses a group of layouts using the specified format. The main loop successively reads and parses a spread (double-page). The parsed results are written into an output file. In the interaction module, the system will use the parsed results and generate an input file

```
2                          // how many pages in the file
20                         // page number of the left
4                          // how many columns on the page
4                          // the number of objects column 0
b82011 hd 82011 1 1 0 8
b8201 ic 82011 1 1 0 7
b8202 da 82011 3 3 0 6
b8203 da 82011 3 3 0 3
1                          // the number of objects column 1
b8204 ic 82011 1 2 1 8
1
b8205 ic 82011 1 2 2 8
5
b82012 hd 82012 1 1 3 8
b8206 ic 82012 1 1 3 7
b8207 ic 82012 1 2 3 6
if821 if 82012 1 2 3 4
df821 df 82012 1 2 3 2

21                         // page number of the right
4                          // how many columns on the page
5
b8208 ic 82012 1 2 0 8
b8209 ic 82012 1 1 0 6
b8210 da 82012 2 2 0 5
df822 df 82012 1 1 0 3
b8211 da 82013 1 2 0 2
3
b8212 ic 82012 1 3 1 8
df823 df 82012 1 1 1 3
b8213 da 82013 1 2 1 2
5
b82013 hd 82013 1 1 2 8
b8214 ic 82013 1 2 2 7
b8215 ic 82013 1 2 2 5
b8216 ic 82013 1 2 2 3
if822 if 82013 1 1 2 1
5
b8217 ic 82013 1 2 3 8
b8218 ic 82013 1 1 3 6
b8219 da 82013 1 3 3 5
df824 df 82013 1 1 3 2
df825 df 82013 1 1 3 1
```

Table 4.1: **An example file of the layout**

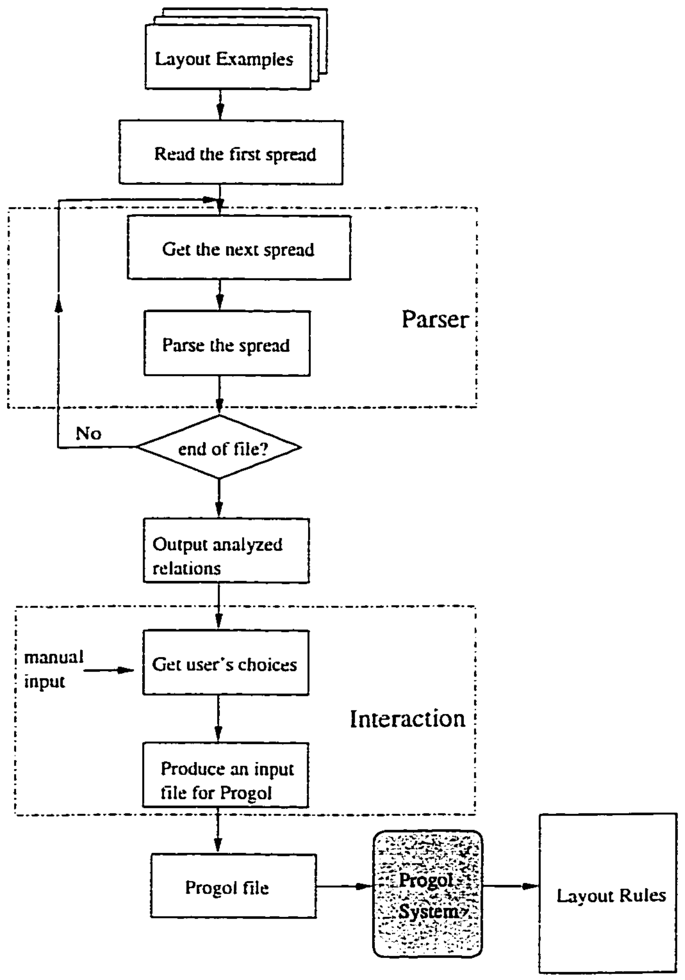


Figure 4.2: General flow chart of the system

for the Progol system according to the user's requests. The input file for Progol contains only some background knowledge and positive or negative examples from the parsed results. We need to manually add some mode declarations so that the Progol system can perform the generalization. The user may set up different mode declarations based on their own interests, in order to explore new and unknown rules.

4.3 The Parser

An input layout file will typically consist of a number of spreads. Each spread contains two pages. On each page, there are a number of columns which hold YP objects (by default, we use four columns). In our design, we have four classes to represent those entities:

Spread A spread is a double-page. It receives and combines the parsed results from its member pages. According to the information from all member pages. it analyzes corresponding attribute relations.

Page A page consists of a number of columns. It receives the parsed results from the columns and records the attribute relations regarding itself.

Column A column is a collection of YP objects. It is responsible for exploring relative relations between objects in itself.

Object An object represents a YP object in the layout. It is identified by several attributes.

4.3.1 The Algorithm for Parsing a Spread

A layout example is usually used to illustrate one or two layout rules. In general. one spread of layout can fully represent the layout designer's intention. Therefore, in our design, we restrict the scope of "orderBefore" to a spread. That means if there is a good example spread of "orderBefore", then the HPDA object must be placed on the right page, and the left page contains those objects which can be placed before the HPDA object. A general description of the algorithm of parsing a spread is shown in Table 4.2.

```
0. fileinfo = a container to save attribute relations from its member pages,
   pages = a vector holding instances of pages of the spread
1. For i= 0 to 1 Do
        parse pages[i], call Page::parse( ).
        add the page's parsed attribute relations into fileinfo.
2. If there is any H(F)PDA object on pages[1]
   begin
        assume the heading of the H(F)PDA object is _h,
        If there is any object of the same heading as _h on pages[0]
        begin
              assume the object's id is oid,
              record the relation of ``orderBefore(oid, h(f)pda, _h)''.
        end.
   end.
```

Table 4.2: **The algorithm for parsing a spread**

4.3.2 The Algorithm for Parsing a Page

We divide the page-parsing process into two parts. The first step is to check the number of different types of objects on the page and record corresponding attribute relations. Then the parser goes through objects column by column in order to get the relative relations between objects. Table 4.3 describes the algorithm.

4.4 Interaction Module

The interaction module serves as a bridge between the parser and the Progol system. Its main function is to communicate with the user and identify those relations in which the user are interested. It then puts those relations into a file following the format which Progol requires. Finally, a manual operation of inserting mode declarations for each Progol file is needed.

4.4.1 Handling of Good Examples

The input to the system includes good or bad examples. In general, an example is good because it complies with one layout rule. The parsed relations from a good example are written into the input file for Progol as *positive examples.* Table 4.4

```
Step 1. Analyze Attribute Relations
1.Repeat
2.    Get a heading h
3.    Check the number of IC, DA, IF, DF, HPDA objects which belong to h
   on the current page
4.    If h does not exist,
        create a new record to save the information.
   Else
        add up the number of that type
5.   Use the above information to get the attribute relations.
6.Until the last object on the page.

Step 2. Analyze Relative Relations
0. cols = total number of columns on the page
columns = a vector holding instances of columns on the page
1. If the page contains an HPDA object
call the function of analyzing HPDA-related relations
2. For each column Do
     get the relation "topCol"
     get the relation "midCol"
     get the relation "touchBot"
     check any adjacent two objects of the relation "downNext" and "touch"
     check the relation underCATEGORY
     check the relation aboveCATEGORY
3. For each column Do
     check every object with objects in other columns about relation "downNext".
4. For each column Do
     check every object with objects in other columns about relation "touch".
5. For i = 0 to cols Do
   For j = i+1 to cols Do
      check each DA object in columns[i] with each DA object in columns[j]
      about the relation of "betterPosition".
6. For i = 0 to cols Do
   For j = i+1 to cols Do
      check each DA object in columns[i] with each DA object in columns[j]
      about the relation of "readingBefore".
```

Table 4.3: **The algorithm of parsing a page**

```
0. Do you want to find a rule about a particular type ?
1. if yes, input the type, assume it is $T$.
2.    For each object of type $T$ in the good examples, Do:
3.         Add a statement of layout(oid, T, P).
4.         output relations it involves.
5. else, are you interested in a particular relation?
6.    if yes, input the relation name, assume it is Name.
7.      For each parsed relation, Do
8.         if its name == Name
7.              output the relation.
9.         endif
10.    endif
11. endif.
```

Table 4.4: **A method of handling good examples**

describes how we add in positive examples into the input file for Progol.

For example, Figure 4.1 shows a good example of how to place DF objects. In Figure 4.1, "df821" touches the bottom of the column. Even though it is not "downNext" to a DA object, it is still considered to be in a valid position. "df822" and "df823" do not touch the bottom of the column. However they are all "downNext" to a DA object. Therefore, they all conform to the rule of "DFsOnlyAtBottomOrUnderDA".

4.4.2 Handling of Bad Examples

A bad example usually relates to a specific layout rule. Since the bad examples are intended to provide negative facts to the Progol system, we have a special method to handle the translation from the parsed relations to the negative examples. Table 4.5 describes how we add in negative examples into the input file for Progol.

Lets use an example to discuss the method. Figure 4.3 gives us a bad example of placing DF objects. All positions that DF objects take are invalid. Therefore, when generating a Progol file, for each DF object, we can write a clause:

```
:-layout(OID,df,P).
```

Here, ":-" means that the clause is negated. "*OID*" is the object ID and "*P*" is the pagenumber.

Followed are the relations that the object involves along with some attribute

```
0. Do you want to find a rule about a particular type ?
1. if yes, input the type, assume it is $T$.
2.    For each object of type $T$ in the negative examples, Do:
3.          Add a statement of :- layout(oid, T, P).
4.          output relations it involves.
5. else, are you interested in a particular relation?
6.    if yes, input the relation name, assume it is Name,
7.      For each parsed relation, Do
8.          if its name == Name, then in the output:
7.              add :- before the relation.
9.          endif
10.     endif
11. endif.
```

Table 4.5: **A method of handling bad examples**

information of the object. It is important to mention that in the bad example, only those DF objects are placed wrong. In the example, object "df820" has relations:

```
downNext(df820, ic, 12).
underICIFs(df820, 12).
midCol(df820, 12).
```

It is obvious that the object "df820" is placed wrong because it does not touch the bottom of the column and is not "downNext" to a DA object.

4.4.3 Background Knowledge

Background knowledge falls into two sets: object definitions and type definitions. In the part of object definitions, each object appearing in the file is defined. For example, `box(b8201).  box(b8202).  box(b8203).`
tells us that object `b8201` `b8202` and `b8203` are instances of variable `box`.

Type definitions, however, are common to all the examples. We can define constants of a type and introduce a new type in this part. The following is a brief example:

```
type(da).  type(ic).  type(df).  type(if).  type(hd).
stype(hpda).  stype(fpda).
```

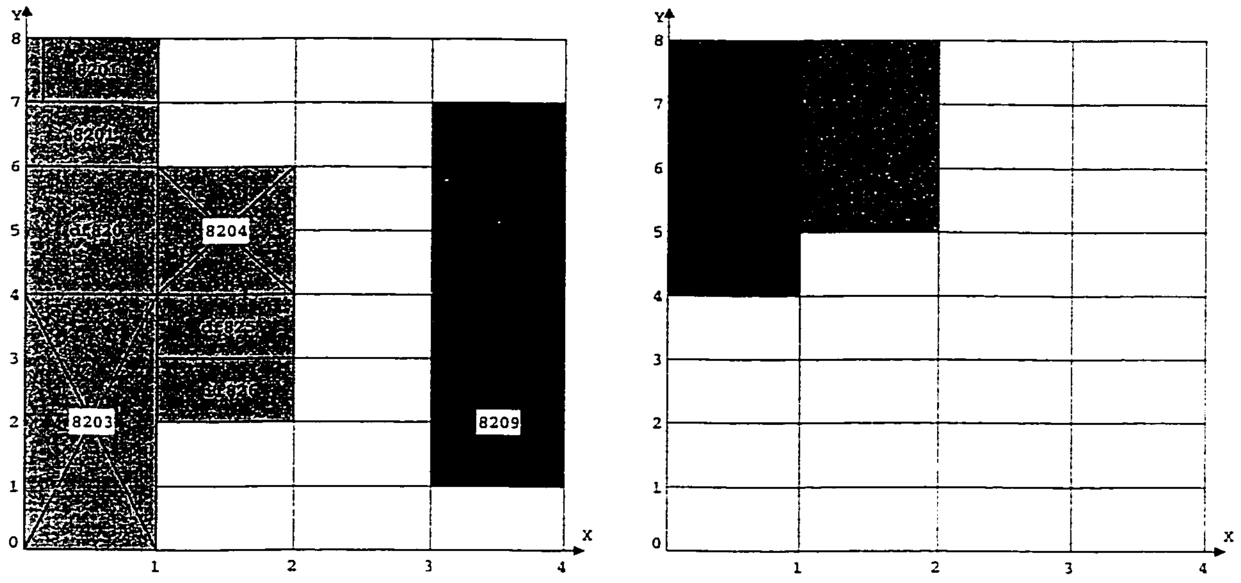


Figure 4.3: **A bad example of placing DF objects**

```
piece(piece(A, T, H, P)) :- box(A), type(T), head(H), int(P).
touch(A,B) :- touch(B,A).
touch(A,C) :- touch(A,B), touch(B,C).
```

In the implementation, we save object definitions part in a background file and append the type definitions as common knowledge.

4.5 Implementation

We use object-oriented design in C++ to implement the parser so that it is easy to adjust the parser to different grid structure of a layout. Another useful tool we rely on is the SGI Standard Template Library. It helps organize efficient and reusable design and avoid redundant work.

The Standard Template Library, or *STL*, is a new C++ library that provides a set of easily composable C++ container classes and generic algorithms ([13]). The container classes include vectors, lists, deques, sets, multisets, maps, multimaps, stacks, queues and priority queues. The generic algorithms include a broad range of fundamental algorithms for the most common kinds of data manipulations, such as searching, sorting, merging, copying, and transforming. The two mechanisms are designed to work together to provide us a wide range of useful functionality. The

```
>read da.good da.bad tmp.res

*********** Starting to generate a PROGOL input file ********
Do you want to generate a file now (y or n)?
y
Please enter the file name without suffix (less than 15 characters)
da.pl

Do you want to look at Relations (1) or Attribute Relations (0)
1
Do you want to find a rule about a particular object type (y or n)?
y
Now please enter the type you are interested in:
da, df, ic, if, hd, hpda, fpda
da
********** End of outputting file [[progol/da.pl ]] *********

*********** Starting to generate a PROGOL input file ********
Do you want to generate a file now (y or n)?
n
================= The END ==================
```

Table 4.6: **A working example**

most important difference between STL and all other C++ container class libraries is that most STL algorithms are generic: they work on a variety of containers and even on ordinary C++ arrays. A key factor in the library design is the consistent use of iterators, which generalize C++ pointers, as intermediaries between algorithms and containers.

The system can be run under a UNIX platform or on a PC under Windows95. We use Progol version 4.2 to do the generalization. Table 4.6 is a working example which shows how the system operates. Bolded fonts are user inputs. In the example, we first read a good and bad example file of placing DA objects. Then we generate a file for Progol by choosing "DA" as the type that we are interested in. The file for Progol is called "da.pl". An extract of the file is shown in Table 4.7.

```
%%%%%%%% Background Knowledge
:- [dagrd]? // note: "dagrd" is a file including each object's definition

%%%%%%%%
type(da). type(ic). type(df). type(if). type(hd).
stype(hpda). stype(fpda).
piece(piece(A, T, H, P)) :- box(A), type(T), head(H), int(P).
touch(A,B) :- touch(B,A).
touch(A,C) :- touch(A,B), touch(B,C).

%%%%%% Positive Examples

layout(b3103, da, 31).
downNext(b3103, ic).
midCol(b3103, 31).
touch(b3103, b3102).
piece(b3102, ic, 80310, 31).
underICIFs(b3103, 31).

layout(b3109, da, 31).
downNext(b3109, if).
touch(b3109, if311).
piece(if311, if, 80312, 31).
touch(b3109, if312).
piece(if312, if, 80312, 31).
touch(b3109, b3115).
piece(b3115, da, 80312, 31).
touchBot(b3109, 31).
underICIFs(b3109, 31).
```

Table 4.7: **An extract of file "da.pl"**

Chapter 5

Tests and Evaluations

5.1 Overview

We evaluate the system by three different tasks. The first task tests if the system can generalize those known layout rules, given good and bad examples. Secondly, suppose a layout designer wants to summarize layout rules of how to place various object types from a set of provided examples; we use the system to construct these interesting rules and see how relevant the rules are. In the last task, we produce some layout examples which imply new layout rules and test if the system can make the generalization and provide us with rules that best reflect user's preferences. All tests are carried out on a SPARCstation20 in the distribution version of Progol4.2. For each task, we examine the system in terms of the runtime taken to construct the rules and number of examples. Moreover, we discuss the experience of using Progol to perform the induction.

5.2 Task 1: Constructing the Known Rules

5.2.1 Overview

In previous chapters we assume we know what layout rules in a real world look like. In the first task, we test if the system can generate those known layout rules from good and bad layout examples. Each good and bad example is chosen to correspond to a particular rule. That is to say, we intend that each rule captures several spreads of good or bad examples. Pictures of all the examples we tested are provided in Appendix B.

We conduct our tests in two stages. At the first stage, we use data from a mixture of good and bad examples. At the second stage, we use good examples and ask Progol to learn rules from positive data only. We will discuss the results from the tests.

5.2.2 A Test Example

We use an example to show how we conduct the test. Suppose the end user draws some good layout examples and bad examples intended to imply the rule "NoLonelyDa". The good examples are presented in Figure 5.1 and Figure 5.2. Bad examples are shown in Figure 5.3. The actual input to the system is file "nolone.good", which saves good layout examples, and file "nolone.bad", which contains bad examples. Table 5.1 shows the process of generating a file for Progol. The bold type fonts are user inputs. In this process, we generated a file called "nolone.pl", as shown in Table 5.2. Note that we wrote all background knowledge into a file called "nolonegrd.pl".

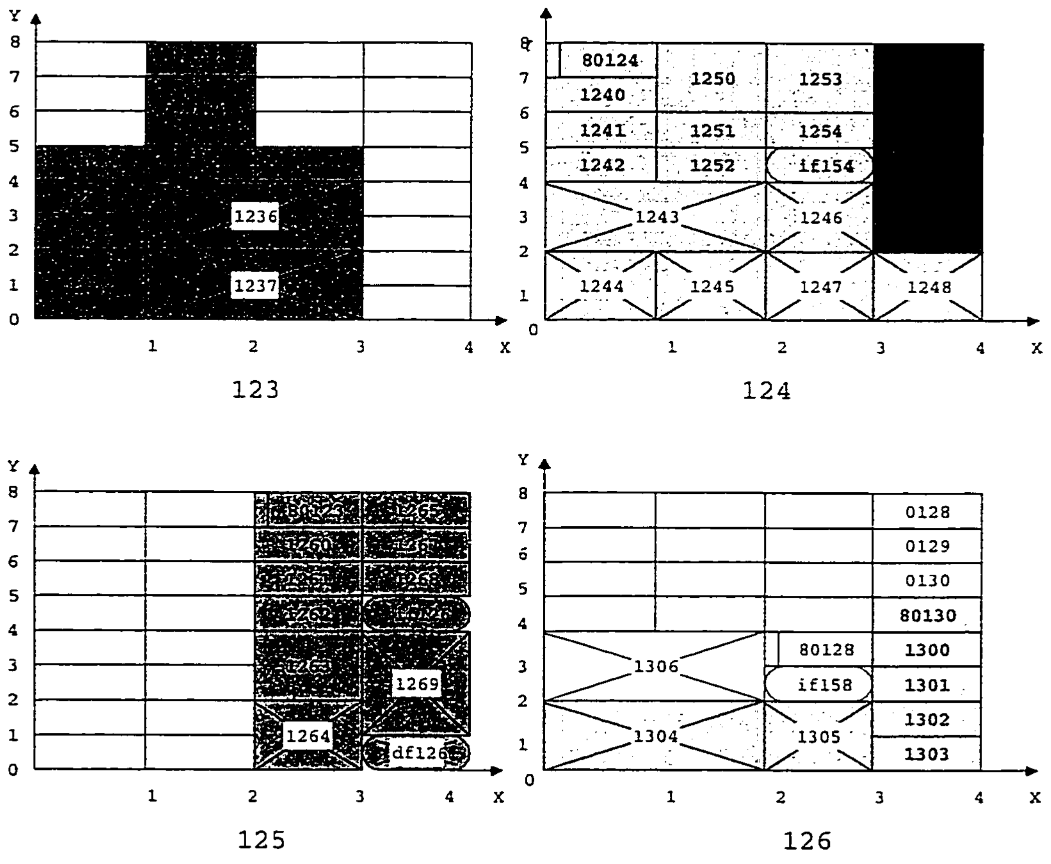


Figure 5.1: **Good examples 1 - rule: NoLonelyDa**

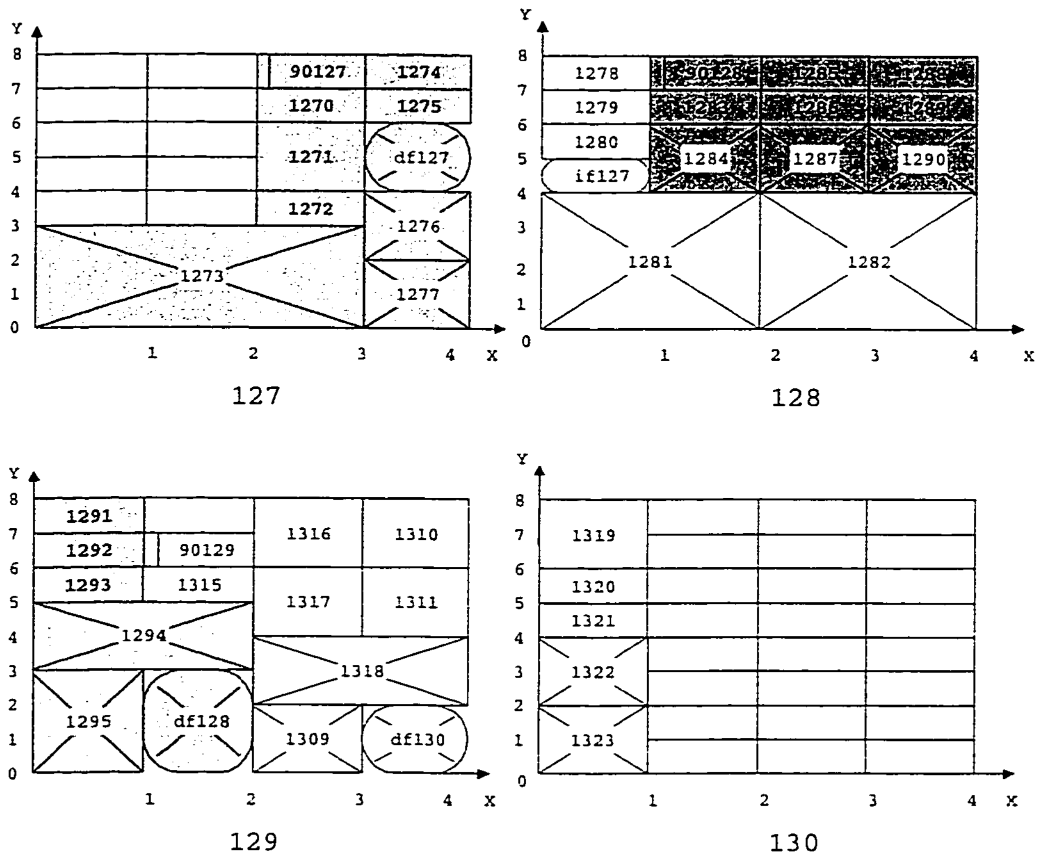


Figure 5.2: **Good examples 2 - rule: NoLonelyDa**

In order to use Progol to perform the generalization, we need to manually add some mode declarations and necessary background knowledge.

We use "das(+list, +head)" to represent the DA distribution of a specific heading on each page containing DAs of that heading. The variable "list" records the number of DAs on each page, and "head" represents the heading.

In this example, the user wants to know what kind of DA distributions on each page is considered to be good for the rule of "NoLonelyDa". Therefore, we added two statements as *modeh*:

:- modeh(1, nolonelyDa(das(+list, +head)))?

:- modeh(1, nolonelyDa(das([+int|+list], +head)))?

The question is: under what constraints, are the numbers in the list considered to be consistent with the rule? The rule "NoLonelyDa" requires that the number of

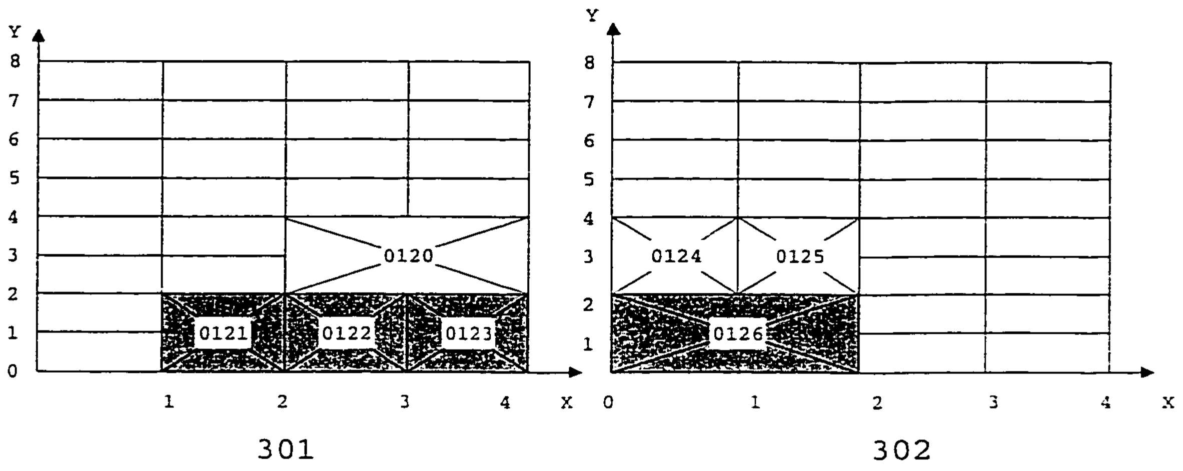


Figure 5.3: **Bad examples - rule: NoLonelyDa**

```
>read nolone.good nolone.bad tmp.res
*** Starting to generate a PROGOL input file ***
Do you want to generate a file now (y or n)?
y
Please enter the file name without suffix (less than 15 characters)
nolone
Do you want to look at Relations (1) or Attribute Relations (0)
0

Please enter two names you want to find out from below:
Use the number to save typing, the numbers must be different
1. icdaSamePage 2. nolonelyDa

2
** End of outputting file [[progol/nolone.pl ]] ***

** Starting to generate a PROGOL input file ***
Do you want to generate a file now (y or n)?

n
================== The END ==================
```

Table 5.1: **Generating a file for rule "NoLonelyDa"**

```
%%%%%%%% Background Knowledge

:- [nolonegrd]?

%%%%%%%%
type(da). type(ic). type(df). type(if). type(hd).
stype(hpda). stype(fpda).
piece(piece(A, T, H, P)) :- box(A), type(T), head(H), int(P).

%%% Positive Examples

nolonelyDa(das([2], h80123)).
nolonelyDa(das([6], h80124)).
nolonelyDa(das([2], h80126)).
nolonelyDa(das([1], h80128)).
nolonelyDa(das([2], h80130)).
nolonelyDa(das([3,2], h90127)).
nolonelyDa(das([3,2], h90128)).
nolonelyDa(das([2,2], h90129)).

%%% Negative Examples
:-nolonelyDa(das([1,2], h12014)).
:-nolonelyDa(das([3,1], h12015)).
```

Table 5.2: **A file for Progol**

DAs on each page must be greater than 1. It implicitly tells us that the selection of *modeb* needs to consider if all numbers in the list is greater than a constant (or an integer). Therefore, we specify the body mode declarations as follows:

```
:- modeb(1, gt(+list, #int))?
:- modeb(1, +list = [ ])?
:- modeb(1, +int > #int)?
%%%%%% Background Knowledge
list([ ]).
list([H|T]) :- int(H), list(T).
gt([ ], _).
gt([H|T], A) :- H > A, gt(T, A).
```

The predicate `gt([H|T], n)` is used to check if each number in the list is greater than a specific number `n`. Its definition is included as background knowledge. In addition, the definition of a list is also included.

Equipped with the above mode declarations and background knowledge, the file shown in Table 5.2 is ready to be input to the Progol system for generalization. The result is shown below:

```
[3 redundant clauses retracted]
nolonelyDa(das([A], B)).
nolonelyDa(das([A|B], C)) :- A > 1, gt(B, 1).
[Total number of clauses = 2]
[Time taken 0.100s]
```

The rule of "NoLonelyDa" allows two situations. One is that all DAs of heading B must be on one page no matter how many DAs the heading has. Another is that if the heading B has more than one DA and there are more one page containing DAs of B, it requires that the number of DAs on each involved page must be greater than 1.

Data set	Predicate	$\|E^+\|$	$\|E^-\|$	$\|B\|$	$\|H\|$	Time(sec)
smda.pl	icdaSamePage	5	2	0	1	0.183
nolone.pl	nolonelyDa	8	2	4	2	0.100
da.pl	layout(A,da,B)	18	3	21	1	0.033
und.pl	underICIFs	21	1	22	2	0.033
nicbf.pl	orderBefore	7	5	12	3	0.067
hpda.pl	layout(A,hpda,B)	12	2	60	3	0.083
above.pl	abovehpda	4	6	10	1	0.017
if1.pl	layout(A,if,B)	7	2	31	2	0.033
df1.pl	layout(A,df,B)	7	4	37	2	0.050
touch.pl	goodTouching	32	1	67	2	7.333
read.pl	readingBefore	20	2	46	2	0.083
rank.pl	betterPosition	19	4	94	2	69.300

Table 5.3: **Progol's runtimes with good and bad examples**

5.2.3 Tests with Positive and Negative Examples

For each rule, we have a file of good examples and a file of bad examples. A list of each source filename and the corresponding file for Progol generated by the system is shown in Appendix C.

Table 5.3 is a summary of runtimes when we learn from those good and bad examples. E^+, E^- represent positive and negative examples respectively. B is the background knowledge, and H is the constructed hypothesis.

As expected, most constructed rules correspond to the ones that we have discussed in Chapter 3. Here, we do not repeat those rules which are the same as in Chapter 3. However, it is important to explain some differences between the rules we got and those expected.

From the file "read.pl", we want to generalize the predicate of "readingBefore". In the beginning of the file, we defined two new predicates and added them as background knowledge:

```
sameHeading(A, B) :- piece(A, _, H, _), piece(B, _, H, _).
headBefore(A, B) :- piece(A, _, H1, _), piece(B, _, H2, _), H1 < H2.
```

The mode declarations are as follows:

```
:- modeh(2, readingBefore(+box, +box))?
:- modeb(1, headBefore(+box, +box))?
:- modeb(1, sameHeading(+box, +box))?
```

The rule we got is:

```
readingBefore(A, B) :- headBefore(A, B).
readingBefore(A, B) :- sameHeading(A, B).
```

The first statement tells us that a DA of the earlier heading is before the DA of the later heading in reading direction, and it explicitly expresses the rule of "ReadingOrder" as discussed in Chapter 3. The second one expresses a complement of the rule. It says that there is no dominating factor controlling reading order if two DAs belong to the same heading.

In the file "rank.pl", the predicate in which we are interested is "betterPosition". The following are the mode declarations:

```
:- modeh(1, betterPosition(+box, +box))?
:- modeb(2, rank(+box, -int))?
:- modeb(2, +int =< -int)?
:- modeb(2, -int =< +int)?
:- modeb(1, sameHeading(+box, +box))?
:- modeb(1, headBefore(+box, +box))?
```

The rules we got are:

```
betterPosition(A,B) :- headBefore(A,B).
betterPosition(A,B) :- rank(A,C),rank(B,D),sameHeading(B,A),D=<E,E=<C.
```

The rules tell us that if two DAs are of different headings, then the DA with earlier heading must take a better position. However, if they are of the same heading, then its rank will determine its grid position. The higher the rank, the better the position.

The above results show that the generated rule provides much more insights to the layout designer of the rule of "readingOrder" and "RankByGridWithinHeading".

In addition, we did some tests of predicates "underICIFs" and "abovehpda" with the following results.

About the rule "NoIcUnderDA", we had two data sets to generalize the rule. From the file "da.pl", we got:

```
layout(A, da, B) :- underICIFs(A, B).
```

From file "und.pl", we got:

```
underICIFs(A, B) :- piece(A, da, C, B).
underICIFs(A, B) :- piece(A, df, C, B).
```

The first rule tells us that a DA should be placed under objects of type IC or IF. The second rule says that DA or DF objects should be placed under objects of type IC or IF.

For rule "NoICsAboveHPDA", we generated two files for Progol. The first one tests the predicate of "layout(A, hpda, B)". It investigates the relative positioning of HPDA objects. The result is:

```
layout(A, hpda, B) :- downNext(A, hpda).
layout(A, hpda, B) :- abovehpda(A, B).
layout(A, hpda, B) :- aboveDADFs(A. B).
```

From another point of view, we want to know which type of object can be placed above an HPDA object. We generated a file called "above.pl", in which we wanted to generalize the predicate of "abovehpda". The result is:

```
abovehpda(A, B) :- piece(A, hpda, C, B).
```

Both rules imply that no IC objects can be placed above HPDA. The first rule enumerates three valid positions that an HPDA could be placed. The second one presents a rule that says only an HPDA object can be placed above another HPDA object.

The above tests demonstrated that Progol is able to generalize those known rules, given good and bad examples. It can even provide us more information about layout if we set up different mode declarations in the files for Progol.

Data set	Predicate	$\|E^+\|$	$\|B\|$	$\|H\|$	Time(sec)
smda.pl	icdaSamePage	5	0	1	0.117
(*)nolonepos.pl	nolonelyDa	11	4	1	0.100
da.pl	layout(A,da,B)	18	18	1	0.100
und.pl	underICIFs	21	21	2	0.117
nicbf.pl	orderBefore	7	7	3	0.067
hpda.pl	layout(A,hpda,B)	12	34	3	0.100
above.pl	abovehpda	4	4	1	0.033
if.pl	layout(A,if,B)	7	23	2	0.050
df.pl	layout(A,df,B)	5	15	1	0.033
touchpos.pl	goodTouching	32	67	2	225.600
readpos.pl	readingBefore	20	42	2	0.383
(*)rkpos1.pl	betterPosition	19	78	2	92.917

Table 5.4: **Progol's runtimes of having positive data only**

5.2.4 Tests with Positive Examples Only

Most ILP systems require both positive and negative examples of ground instances of a predicate. Progol, however, provides a mechanism which can learn from positive only data with low expected error [17]. Since explicit negative examples of a predicate are not always readily available, it might be interesting to investigate the results from Progol when given solely good examples.

We used the files of good examples in Table C.1 as input to the system and generated files for Progol. Table 5.4 lists the results from learning positive data only.

Rules generated from files without * in Table 5.4 are the same as rules we got from using positive and negative examples. However, the required runtime of "touchpos.pl" is about 32 times of the file "touch.pl" in Table 5.3. It is mainly because we have such recursive relations: `''touch(A, B) :- touch(B, A)''` `''touch(A, B) :- touch(A, C), touch(C, D)''`.

Progol can generalize rules for files with * in Table 5.4, but the results do not express the rules the way that we expected. Table 5.5 lists the rules we got from those files. Rule 1 tells us if the number of DAs on the first page is greater than 0, then it complies with the rule of "NoLonelyDa". From rule 2 we do not know under what constraints that one DA should be placed at a better position than another. The two unexpected rules resulted from the small number of examples, which made

Data set	Result
nolonepos	1. nolonelyDa(das([A|B],C)) :- A > 0.
rkposl	2. betterPosition(A,B) :- rank(A,C), rank(B,D).

Table 5.5: **Tests using positive only: a list of unexpected rules**

the rules over-general.

The file "nolonepos.pl" contains only 11 positive examples. If we had a total of 52 examples (refer to file "t2.pl"), we could have the following rule:

```
nolonelyDa(das([A], B)) :- A>0.
nolonelyDa(das(A, B)) :- gt(A, 1).
```

The file "rkposl.pl" contains only 20 examples. Similarly, if we added 52 more examples (refer to "rkpos2.pl"), we got:

```
betterPosition(A, B) :- headBefore(A, B).
betterPosition(A, B) :- rank(A, C), rank(B, D), D=<E, E=<C.
```

The experiments show that Progol needs sufficient number of examples when learning from positive only data, especially for multiple clause hypothesis. This can also be confirmed from the theoretical framework of Progol4.2, which involves an algorithm for learning from positive data only.

Progol4.2 uses the Bayes' function f_m to guide the search. Suppose H represents hypothesis, m is the number of examples, and $g(H)$ is the generality of H. The Bayes' function f_m is:

$$f_m(H) = c.2^{-|H|}(1 - g(H))^m \tag{5.1}$$

We can see that f_m trades off the complexity of a hypothesis against its generality. When constructing each clause, Progol "carries out an admissible search which optimsies a global estimate of f_m for the complete theory containing the clause under construction" ([17]). The basis behind the global estimate is as follows: Suppose a clause C_i has been constructed as the ith clause of an overall theory $H_n = C_1, \cdots, C_n$. It is found that when m is large it is possible to estimate both $sz(H_n)$ and $g(H_n)$.

```
%%% Positive Examples

nolonelyDa(das([2], h80123)).
nolonelyDa(das([6], h80124)).
nolonelyDa(das([2], h80126)).
nolonelyDa(das([1], h80128)).
nolonelyDa(das([2], h80130)).
nolonelyDa(das([3,2], h90127)).
nolonelyDa(das([3,2], h90128)).
nolonelyDa(das([3,2], h90129)).

%%% Negative Examples
:-nolonelyDa(das([1,2], h12014)).
:-nolonelyDa(das([3,1], h12015)).
```

Table 5.6: **Another file about rule "NoLonelyDa"**

Therefore, it is possible to maximise an estimate of $f_m(H_n)$ during the construction of each of the clauses.

5.2.5 Discussion

When investigating the performance of the system, we notice that the number of examples affects the construction of rules. Especially for *Progol*, a rule involving integers that is not completely determined by a single example can be difficult to learn. For example, consider the following test. On page 129 of Figure 5.2, we have two DAs of heading 90129: *1318* and *1309*. If we modified the picture by replacing the DF object *df130* with another DA object 1312 of heading 90129, then there will be three DA objects on this page.

After parsing, the system produced a group of positive examples, shown in Table 5.6. Note that the line with bold fonts are different from that of Table 5.2.

With Table 5.6 as input, Progol produced:

```
nolonelyDa(das([A], B)).
nolonelyDa(das([A|B], C)) :- A> 2, gt(B, 1).
```

This rule would require that on the first page the number of DAs of heading C must be greater than two, and the subsequent pages must each have more than one DA of heading C. The reason why we got such an unexpected rule is that the examples did not provide enough information for induction.

5.3 Task 2: Finding Rules for Placing Each Object Type

5.3.1 Overview

In task 1, we used Progol to generalize each known rule. However, we did not comprehensively consider all relations which must be obeyed for an object type. For example, in task 1, we have a rule for IF's position in one column:

```
layout(A, if, B) :- touchBot(A, B).
layout(A, if, B) :- midCol(A, B).
```

The rule does not tell us that an IF object must obey the rule of about being under objects of DA or DF. In order to obtain a complete set of constraints for placing each type, we need to parse each good and bad example, and save all relations that each object of that particular type participates in. Then, we check if Progol can make a compression and construct new rules.

We consider five types of YP objects: DA, DF, IC, IF and HPDA. For each type, we have good examples (i.e. file "da.good") and bad examples (i.e. file "da.bad"). An important assumption is that every object of that type in the bad examples must be incorrectly placed according to the user's intention.

Figure 5.4 shows the picture of the file "da.bad". In Figure 5.4, every DA object is incorrectly placed according to the user's preference. An example is object 1011. It is incorrectly placed because it is above an IC object 1012. Another bad example is object 1021 on page 312. Although it has a relation of "underICIFs", it does not touch any IC(F) object of its heading.

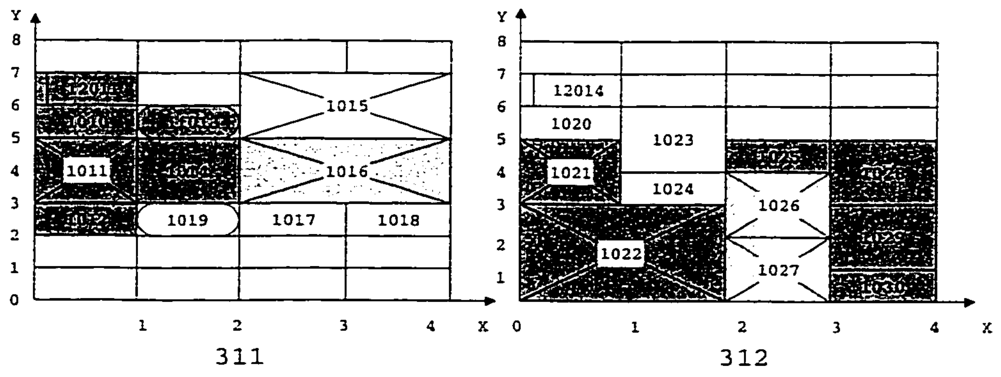


Figure 5.4: **A bad example of placing DAs**

5.3.2 Analysis

Since mode declarations are at the heart of Progol's method of generalizing examples, we discuss how we set up the mode declarations. Then follows the discussion of the results.

In the layout, each good example is considered to be in a valid position, and each bad example is considered to be in a invalid position. All relations in which each example object participates are saved as background knowledge. The *modeh* should be:

:- modeh(1, layout(+box, #type, +int))?

There are two factors that determine the *modeb* declarations for Progol. One is to observe relations that almost all objects participate in. For example, if we look at all DA objects and find out that almost all DA objects participate in the relation of "underICIFs", then we put this into the *modeb*. The second is to select relations of most interest to the user. For example, if the user is interested in generalizations of specific relations, "topCol", he may put it into the *modeb*. The files in task 2 are presented in Appendix C.

Table 5.7 shows the resulting rules computed by Progol. Compared with the rules we got before, we can see that these rules provide new and complete constraints for placing an object type. For instance, a DA object must be under objects of IC or IF, and it must touch either an IC or IF of the same heading. There are two cases in which DF object is placed validly. One is that it touches the bottom of the column,

Data Set	Rule
da2.pl	layout(A, da, B) :- underICIFs(A, B), touch(A, C), piece(C, ic, D, B). layout(A, da, B) :- underICIFs(A, B), touch(A, C), piece(C, if, D, B).
df2.pl	layout(A, df, B) :- touchBot(A, B). layout(A, df, B) :- downNext(A, da), underICIFs(A, B).
if2.pl	layout(A, if, B) :- touchBot(A, B). layout(A, if, B) :- aboveDADFs(A, B), midCol(A, B).
ic2.pl	layout(A, ic, B) :- aboveDADFs(A, B).
hpda2.pl	layout(A, hpda, B) :- downNext(A, hpda). layout(A, hpda, B) :- abovehpda(A, B). layout(A, hpda, B) :- aboveDADFs(A, B).

Table 5.7: **Task 2: rules generated by Progol**

the other is that it is under and adjacent to a DA and also under objects of IC(F). For an IF object, it is valid if it touches the bottom of a column. Otherwise, it can be placed in the middle of a column, and must be above objects of DA(F). The constraint for IC is simple. It requires that an IC must be always above objects of DA(F).

5.4 Task 3: Finding New Rules

The system is designed to be able to construct rules based on user-provided examples. In task 3, we draw some layouts which imply different rules from those we tested before, and see what the system generates.

First, we have some layout examples which illustrate a new definition of rule "NoLonelyDA". Given the good examples shown in Figure 5.5 and the bad examples shown in Figure 5.6, will the system be able to generate a rule of what DA distribution on each page is acceptable?

The result is positive. If a heading Hg has more than one DA, then it requires that the first page having DAs of that heading must also have at least two DAs. However,

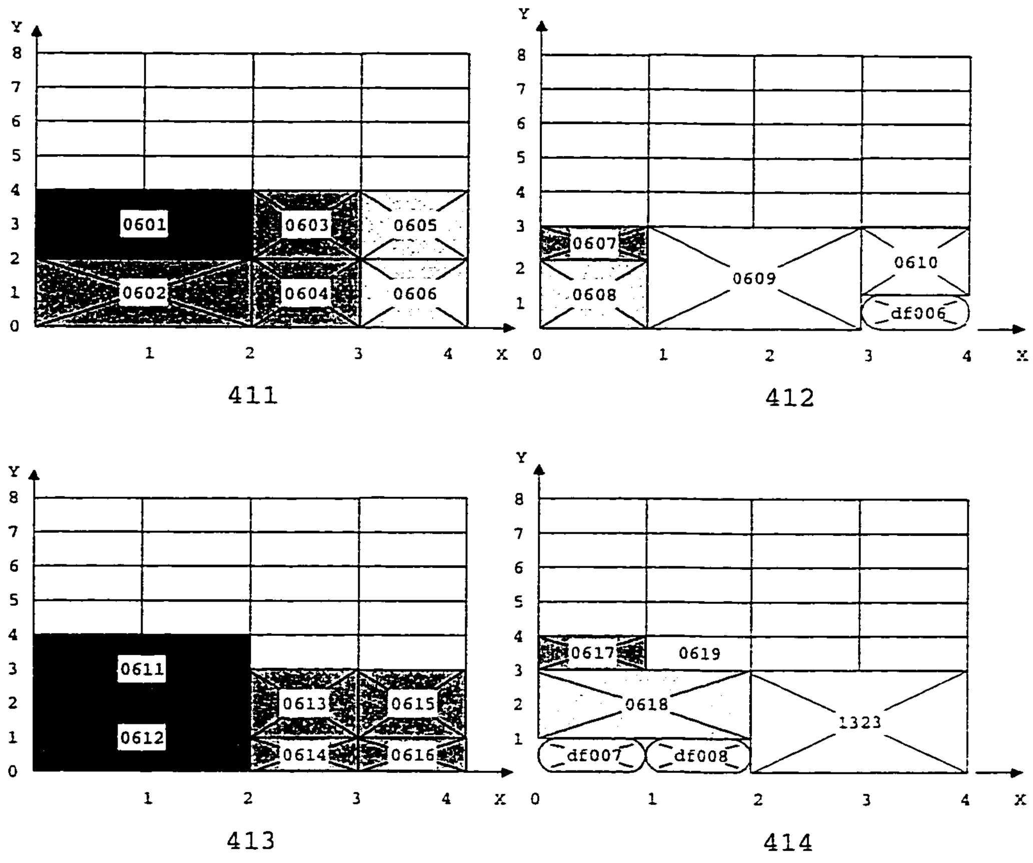


Figure 5.5: **A good example of new rule: NoLonelyDa**

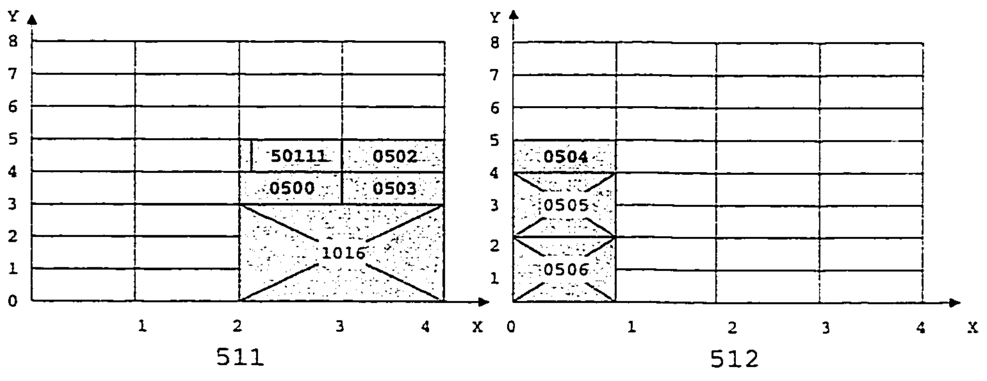


Figure 5.6: **A bad example of new rule : NoLonelyDa**

it does not insist that each page having DAs of heading Hg must have more than one DA of heading Hg.

nolonelyDa(das([A], B)).

nolonelyDa(das([A|B], C)) :- A> 1.

Now, suppose we have spreads of good examples of placing various object types. The good examples are presented by two files: "task3.good" and "thpda.good". We show the pictures of file "task3.good" in Figure 5.7, and file "thpda.good" is presented in Figure 5.8.

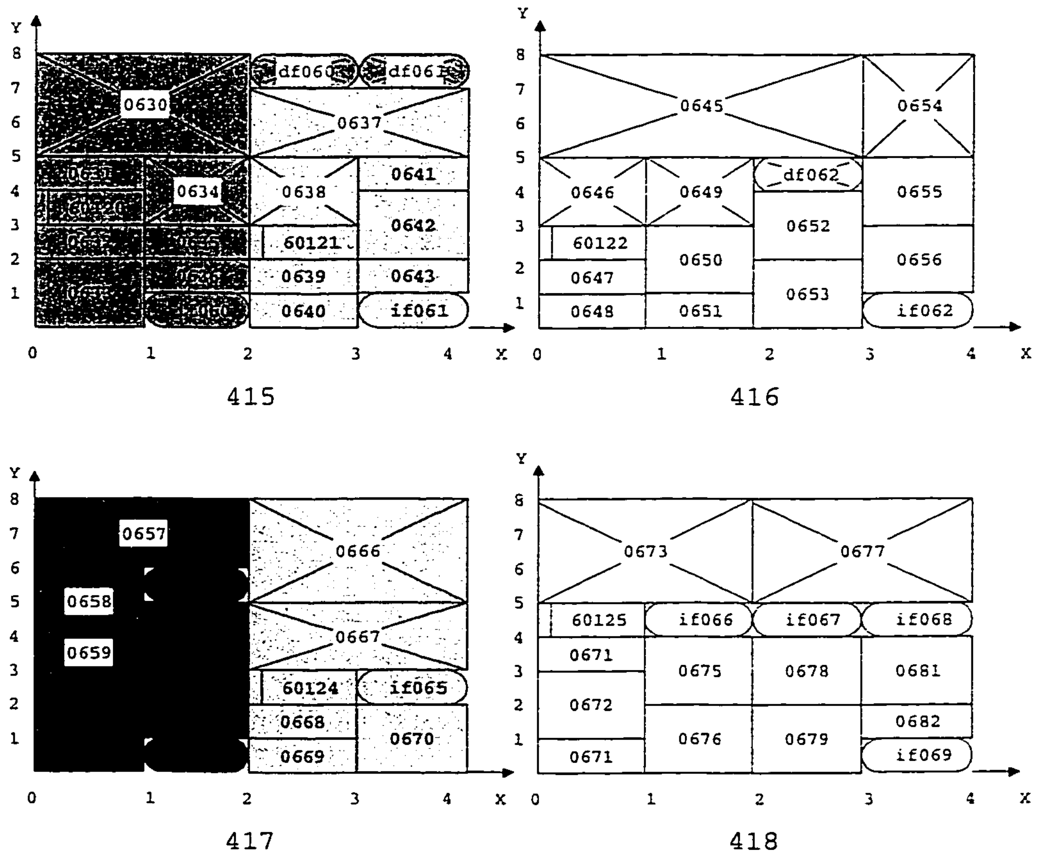


Figure 5.7: **Good examples of placing DA(F) and IC(F) - file "task3.good"**

From the pictures, we can see that there are different rules for placing each object type from what we discussed before. The test is to check if the system can generate some new rules for placing each object type. In order to achieve this, we need bad

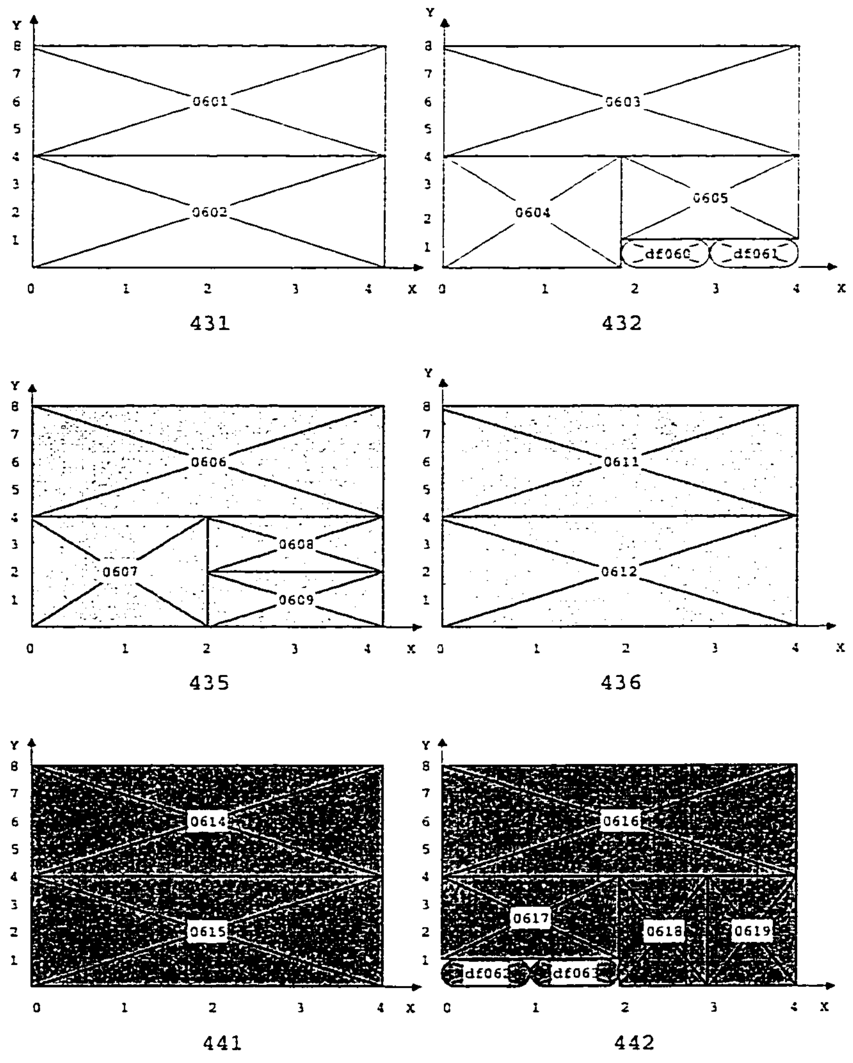


Figure 5.8: **Good examples of placing HPDA - file "thpda.good"**

examples of placing each object type. There are 5 files of bad examples. Each file has an example of one object type. File "t-da.bad" means that each DA in the example is placed wrong and file "t-ic.bad" means that it contains bad examples of placing IC objects; the same rule applies to other three files. All bad examples are shown in Figure 5.9, Figure 5.10, Figure 5.11, Figure 5.12, and Figure 5.13.

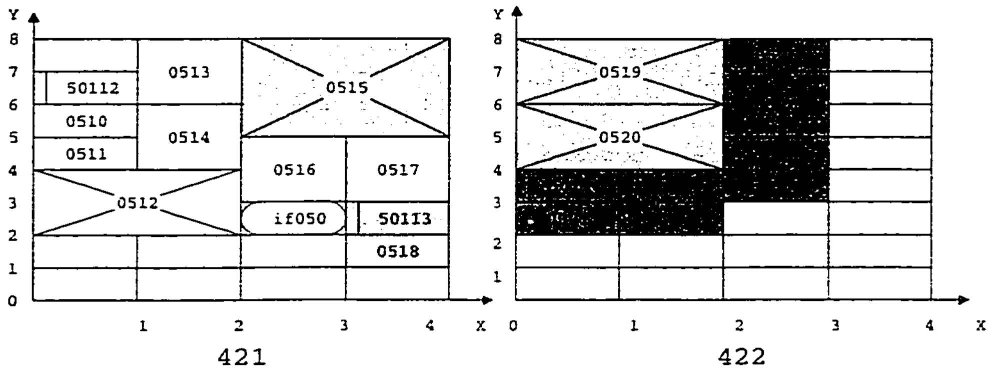


Figure 5.9: **Bad examples of placing DA - file "t-da.bad"**

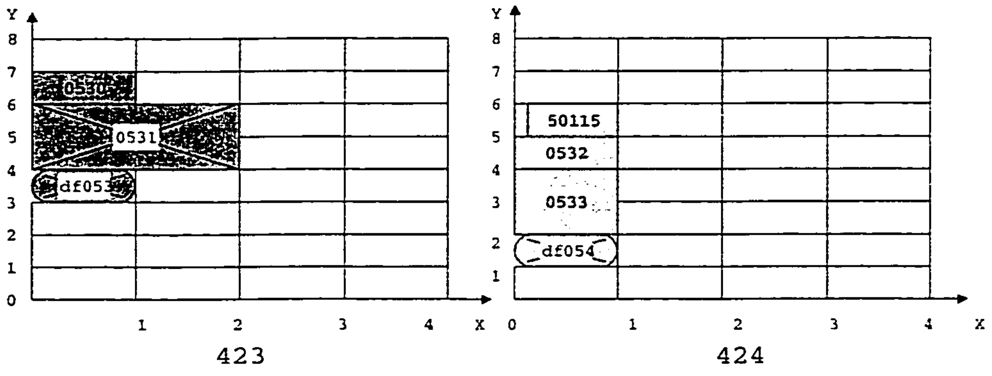


Figure 5.10: **Bad examples of placing DF - file "t-df.bad"**

Table 5.8 provides the Progol runtime for each file. The specification of mode declarations in each file follows the same criteria which we discussed in Task 2.

The results provide some new rules for placing each object type. We listed all rules in Table 5.9. For DAs, we need to place it above any object of IC(F), and it must touch an object of IC(F) of its heading. The constraints for DF are fewer. As

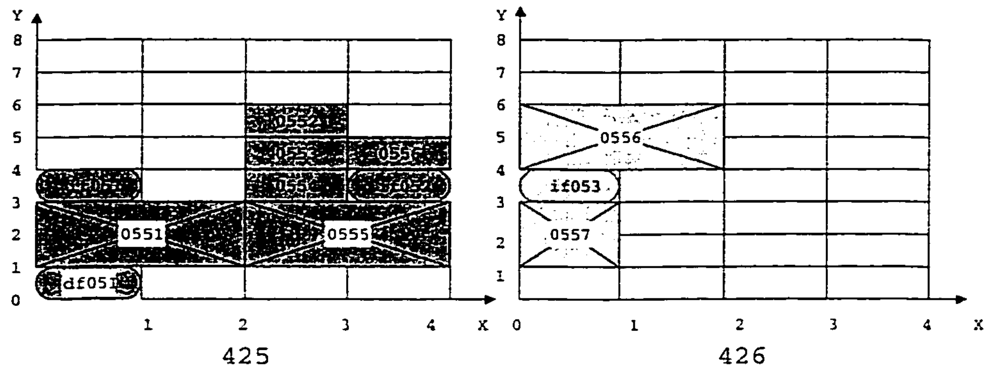


Figure 5.11: **Bad examples of placing IF - file "t-if.bad"**

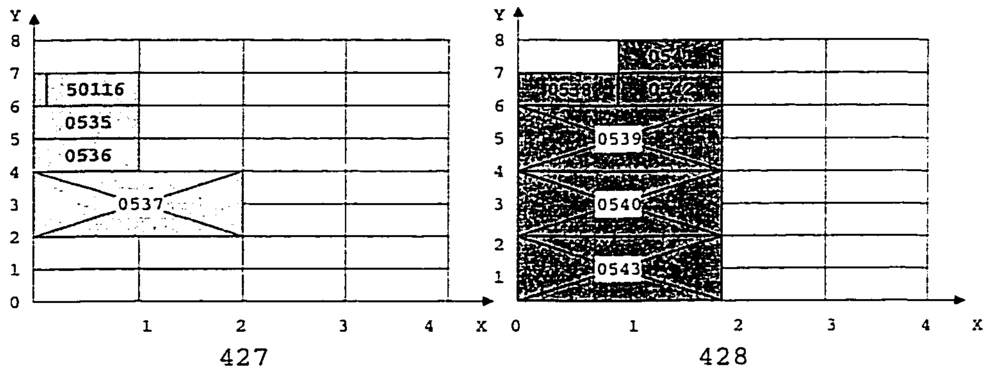


Figure 5.12: **Bad examples of placing IC - file "t-ic.bad"**

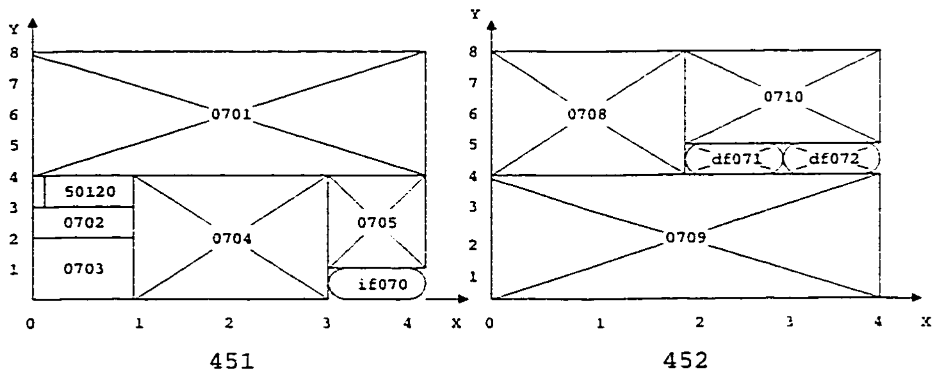


Figure 5.13: **Bad examples of placing HPDA - file "t-hpda.bad"**

Data set	Predicate	$\lvert E^+\rvert$	$\lvert E^-\rvert$	$\lvert B\rvert$	$\lvert H\rvert$	Time(sec)
t-da.pl	layout(A, da, B)	15	4	88	2	21.900
t-df.pl	layout(A, df, B)	3	2	14	1	0.050
t-if.pl	layout(A, if, B)	9	3	39	1	0.033
t-ic.pl	layout(A, ic, B)	34	5	126	1	0.050
t-hpda.pl	layout(A, hpda, B)	9	2	34	3	0.067

Table 5.8: **Task 3: Progol's runtimes**

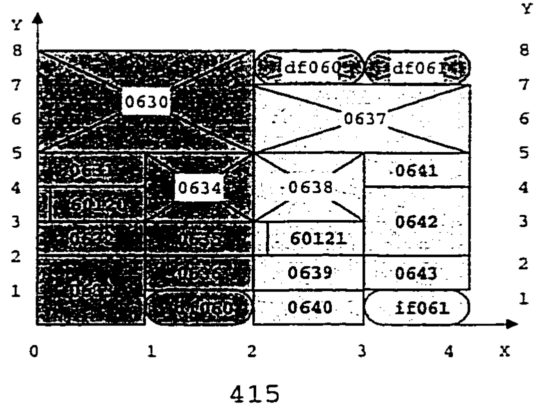
Data Set	Rule
t-da.pl	layout(A, da, B) :- aboveICIFs(A,B), touch(A,C), piece(C,ic,D,B). layout(A, da, B) :- aboveICIFs(A,B), touch(A,C), piece(C,if,D,B).
t-df.pl	layout(A, df, B) :- aboveICIFs(A, B).
t-if.pl	layout(A, if, B) :- underDADFs(A, B).
t-ic.pl	layout(A, ic, B) :- underDADFs(A, B).
t-hpda.pl	layout(A, hpda, B) :- downNext(A, hpda). layout(A, hpda, B) :- abovehpda(A, B). layout(A, hpda, B) :- aboveDADFs(A, B).

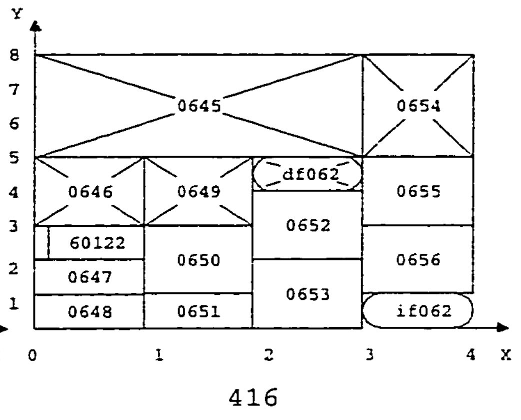
Table 5.9: **Task 3: rules generated by Progol**

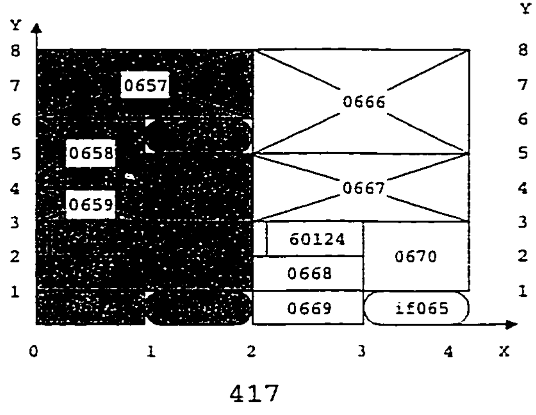
long as a DF is placed above objects of IC(F), it is considered to be good. For IC and IF, it is valid if they are placed under DA(F). The rule of placing HPDA is the same as before. It is either above another HPDA or under another HPDA, or it can be above objects of DA(F). In other words, HPDA cannot be above objects of IC(F).

5.4.1 Analysis

The success of Task 3 shows that our system is able to find new rules given good and bad examples. From the experiments, we know that the quality of a rule is affected by the number of positive and negative examples. Empirically we observe that, for a rule which has more than one clauses as rule body, for example, $H \leftarrow B_1, \cdots, B_n$, each condition needs at least 6 positive examples.







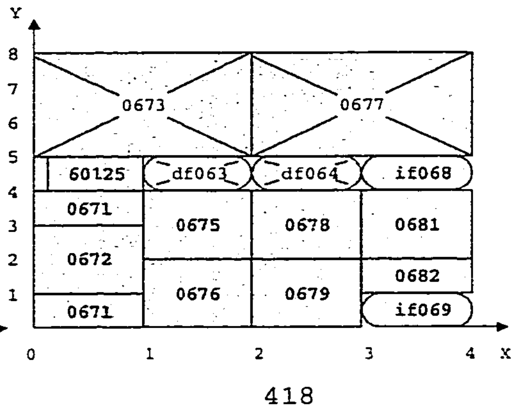


Figure 5.14: **Examples in a test file "test.good"**

For example, in Figure 5.7 , we have 9 good examples of a DA which touches an IC of its heading, and 6 good examples of a DA which touches an IF. If we change the layout and reduce the number of DAs which touch IF, we get the updated layout shown in Figure 5.14. On page 417, we put IC object 0670 above *if*065, and two IF objects (if066, if067) on page 418 are replaced by two DF objects (df063, df064). In this case, there are only two DA objects, b0657 and b0677, which touch IF objects. The test produces a result of no generalization, as shown below:

```
[No compression]

layout(b0673, da, 418).

layout(b0677, da, 418).

layout(A, da, B) :- aboveICIFs(A, B), touch(A, C), piece(C, ic, D, B).

[Total number of clauses = 3]
```

From all the positive examples, Progol can get only the last clause, which covers most DAs which touch IC of the same heading. However, it cannot explain the two objects, b0673 and b0677. The two objects are regarded as good examples, however, they do not touch any IC object.

Chapter 6

Summary and Future Work

6.1 Conclusions

ILP systems have been successfully applied to various domains. [9] shows an experimental study of automatic knowledge acquisition of expertise in an e-mail classification expert system. A data mining system DB-Amp is used to extract relevant knowledge from a database, and Progol is then used to infer a set of e-mail classification rules. The system succeeded in applying the rules to an operational expert system and in showing its usefulness in a real business environment. Another ILP system, Golem, has been applied to the problem of finite element mesh design [16]. Compared with other techniques of machine learning, ILP techniques have the advantages of outputting easily understandable rules and the ability to make use of background knowledge in different application domains.

The constraint-based approach in YP layout design is useful because of its flexibility, and its declarative representation of design knowledge. Consequently, obtaining layout constraints from a group of layout examples is an interesting problem and worthy of study. This dissertation has studied the problem of inducing layout rules using an ILP system, Progol. There is no previous work of using ILP techniques in inducing YP layout rules. Our work is the first application of Progol to the layout rules learning problem.

We concentrate on the following issues:

1. how to identify necessary background knowledge (BK) and how to define predicates constituting the BK;

2. given a set of layout examples, how to design a target concept representation to be learned by Progol.

We studied layout rules in a real world and defined a set of relations (predicates) which capture the attributes of real-world layout rules. The definition of the target representation is determined by its "interestingness" to a layout designer and its clarity in helping to express the rules.

Second, we developed a parser which is able to analyze those relations from layout examples.

Our prototype system generates input files for Progol based on user's instructions. The definition of background predicates and mode declarations are added manually into the files for Progol.

We conducted experiments on three different tasks. Given good and bad examples, Progol is able to produce those known layout rules. However, our test with positive only data shows that Progol cannot provide accurate rules. In addition, we did tests on some good and bad examples of placing each YP object type. The system demonstrated its ability to induce some new and potentially useful rules, such as, how to place objects of each type. Furthermore, we applied the system to a set of layout examples which imply different styles of layout. The results confirms that the system is able to produce useful layout rules. Overall, it turns out that the quality of rules is affected by the number of provided examples.

6.2 Future Work

There are two directions to extend our work.

From the theoretical point of view, we plan to investigate which factors are important to influence the induction of "better" layout rules. We need to consider the following aspects: First, for ILP systems, we can specify hypotheses in order to constrain the search and improve the efficiency. How would the setup of possible hypotheses affect the resulting rules? In this case, we can extend our work to use other ILP systems, such as, GOLEM or FOIL, and see if we can get more expressive results without losing efficiency.

Second, the predicates constituting BK are crucial to the induction in ILP. Currently, we define a small set of predicates which are able to generate those known rules. However, it will be insufficient to a more complex layout. One example is a page full of DAs. How can we induce the constraints in such a case? These questions can be further topics for study.

From the practical point of view, we need to focus on the following issues:

- Evaluation

  The essential goal of layout rules is to help layout designers in YP design. To evaluate its correctness, we need to apply the rules into the YPPS system and find out if the generated layout is acceptable.

- Translation of pictures

  Our system assumes a fixed file format of input pictures. Further work needs to be done to build a translator which can translate the 2D pictures into the file format which our system can accept.

- Identification of Examples

  The identification of positive and negative examples is still a problem. In the system, we assume each object in good example file as good, and each object in bad example file as bad. An enhancement to the system would be an GUI in which the user can specify good and bad layout examples, as well as to which rule they are justified as good or bad.

- Quality Constraints

  We leave the study to quality constraints as the future work. There are some issues that need to be considered: a representation scheme in order to represent the optimization quality rules and new relations that are used to distinguish layouts of different orders with respect to a rule.

Bibliography

[1] Ramesh Johari, Joe Marks, Ali Partovi, Stuart Shieber, "Automatic Yellow-Pages Pagination and Layout",*Journal of Heuristics*, 2:1997, pp321-342.

[2] Chew Hong-Gian,Moung Liang, et.al, "ALEXIS: An Intelligent Layout Tool for Publishing", *Proceedings of the Sixth Annual Conference on Innovative Applications of Artificial Intelligence*, Seattle, WA, 1994, pp41-47.

[3] Graf W.H., S.Neurohr, and R.G.Goebel,"YPPS-A Constraint-Based Tool for the Pagination of Yellow-Page Directories",*Technical Report*, German Research Center for Artificial Intelligence(DFKI) GmbH.

[4] Graf W.H., S.Neurohr, R.G.Goebel, "Experience in Integrating AI and Constraint Programming Methods for Automated Yellow Pages Layout",*Technical Report*, German Research Center for Artificial Intelligence(DFKI) GmbH.

[5] Graf W.H., "Constraint-Based Graphical Layout of Multimodal Presentations",*Technical Report*, German Research Center for Artificial Intelligence(DFKI) GmbH.

[6] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods", *Journal of Logic Programming*, 19,20:629–679, 1994.

[7] S. Muggleton, "Inductive logic Programming", *New Generation Computing*, 8(4)1991, pp295–318.

[8] S. Roberts, "An Introduction to Progol",*Technical Report*, Oxford University Computing Laboratory, January 1997.

[9] K. Shimazu, K. Furukawa, "Knowledge Discovery in Database by PROGOL - Design, Implementation and its Application to Expert System Building", SIG-FAI-9601-14, pp88 -104.

[10] S. Muggleton (Ed.), "Inductive Logic Programming", Academic Press, 1992.

[11] Philip D. Laird, *Learning from Good and Bad Data*, Kluwer Academic Publishers, 1988.

[12] G.D. Plotkin, "A Note on Inductive Generalization", in B. Meltzer and D. Michie, editors, *Machine Intelligence*, vol.5, Elsevier NorthHolland, New York, 1970.

[13] "Standard Template Library Programmer's Guide", *http://www.sgi.com/Technology/STL/index.html.*

[14] S. Muggleton, R. King, and M. Sternberg, "Protein Secondary Structure Prediction using Logic-based machine learning", *Protein Engineering*, 5(7):647-657, 1992.

[15] N. Lavrač and S. Džeroski, "Inductive Logic Programming, Techniques and Applications", Ellis Horwood, 1994.

[16] B. Dolšak and S.Muggleton, "The Application of Inductive Logic Programming to Finite-Element Mesh Design", In Muggleton, S., editor, *Inductive Logic Programming*, pages 453-472. Academic Press, London.

[17] S.Muggleton, "Learning from Positive Data", In Muggleton, S., editor, *Inductive Logic Programming, 6th International Workshop*, pp358-376, Springer.

[18] S.Muggleton, "Inverse Entailment and Progol", *New Generation Computing*, Vol.13, 1995, pp245-286.

[19] I. Bratko and S. Muggleton, "Applications of Inductive Logic Programming", *Communications of the ACM*, Vol.38, No. 11, Novermber 1995, pp65-70.

[20] K. Miyashita, S. Matsuoka, S. Takahashi, and A. Yonezawa, "Interactive Generation of Graphical User Interfaces by Multiple Visual Examples", *UIST'94*, November 2-4,1994, pp85-94.

[21] S. Takahashi, S. Matsuoka, A. Yonezawa, and T. Kamada, "A General Framework for Bi-Directional Translation between Abstract and Pictorial Data", *UIST'91*, November 11-13,1991, pp165-174.

[22] K. Miyashita, S. Matsuoka, et.al, "Declarative Programming of Graphical Interfaces by Visual Examples", *UIST'92*, November 15-18,1992, pp107-116.

[23] J.R. Quinlan, "Learning Logical Definitions from Relations", *Machine Learning*, 5, 1990, pp239-266.

[24] S. Muggleton, "Inductive Logic Programming: derivations, successes and shortcomings", *SIGART Bulletin*, Vol.5 No. 1,1994, pp5-11.

[25] S. Muggleton and C. Feng, "Efficient Induction of Logic Programs", S. Muggleton (Ed.), *Inductive Logic Programming*, Academic Press, 1992, pp281-298.

[26] R. King, S. Muggleton, R. Lewis, and M. Sternberg, "Drug Design by Machine Learning: The Use of Inductive Logic Programming to Model the Structure-activity Relationships of Trimethoprim Analogues Binding to Dihydrofolate Reductase", *Proceedings of the National Academy of Sciences*, 89(23),1992.

[27] J. Cussens, D. Page, S. Muggleton and A. Srinivasan, "Using Inductive Logic Programming for Natural Language Processing", *Technical Report*, Oxford University Computing Laboratory, February, 1997.

[28] J. Cussens, "Part-of-speech Disambiuation Using ILP", *Technical Report*, Oxford University Computing Laboratory, 1996.

[29] J.R. Quinlan, "Induction of Decision Trees", *Machine Learning*, Vol. 1, 1986, pp81-106.

[30] R.S. Michalski, et.al, "The Multipurpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains", *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA: Morgan Kaufmann. 1986, pp1041-1045.

Appendix A

Definition of Terms

Here, we list terms that we used in this dissertation by giving its name, definition, and the page number related to it.

DA Display advertisement object (page 1, 21)

DADFs A category which includes display ad and display filler object (page 30)

DF Display filler object (page 1, 21)

DFsOnlyAtBottomOrUnderDA Display filler must only be placed under a DA or on positions that touch the bottom. (page 26)

FPDA Full page display ad object (page 21)

heading All YP objects are classified by a heading object (page 1,21)

HPDA Half page display ad object. (page 21)

IC Incolumn text stream object (page 1,21)

ICIFs A category which includes incolumn text and incolumn filler object (page 30)

IF Incolumn filler object (page 1,21)

ILP Inductive logic programming (page 5)

NoICsAboveHPDA Above a Half Page Display Ads (HPDA's) there must be no IC. (page 24)

NoICsBeforeHalfPageOrFullPageDA When a heading contains a DA that has the size of a half page (HPDA) or a full page (FPDA), then these ads must be the first placed objects of this heading. (page 26)

NoIcUnderDA There must be no ICs under Das. (page 24)

NoIFsAtTopOfPage Incolumn filler must not be placed at the top of a column. A column must either start with an Incolumn or Display Ad objects. (page 26)

NolonelyDA If a DA is smaller than a half page and if it is not the only DA of it's heading, it must not be placed as the only DA of its heading on a page. (page 24)

rank We use "rank" to order each DA object based on its size. (page 22)

RankByGridWithinHeading When there are two DAs of the same heading on the same scope, then the DA with the better rank must be on a better position than the DA with the worse rank. The quality of a position P is reflected by the grid weight of the grid field in the upper-outer corner of the DA at position P. (page 27)

ReadingOrder If there are two DAs of two different headings on a scope, then the DA of the earlier heading must be placed before the DA of the later heading in reading direction. (page 29)

scope page or spread, depending on the current layout scope. (page 22)

SmallDAsAndICsStartOnSamePage Consider the first page where ads of heading h are placed. If this page contains a small DA, then there must be placed some ICs of that heading. (page 23)

spread double page (page 1)

Touching Each DA of a heading must touch transitively an IC of its heading. In other words: A DA touches, if and only if it touches an IC of it's heading or another DA which touches. (page 26)

YP telephone book Yellow Pages^{TM} (page 1)

Appendix B

Pictures of Examples

B.1 Good Examples

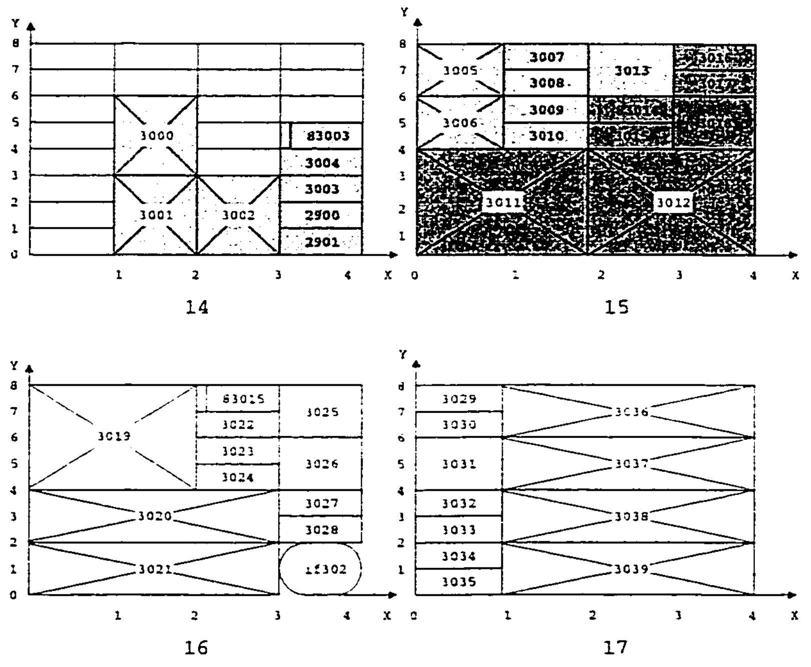


Figure B.1: Good examples of rule:SmallDAsAndICsStartOnSamePage - file: smda1.good

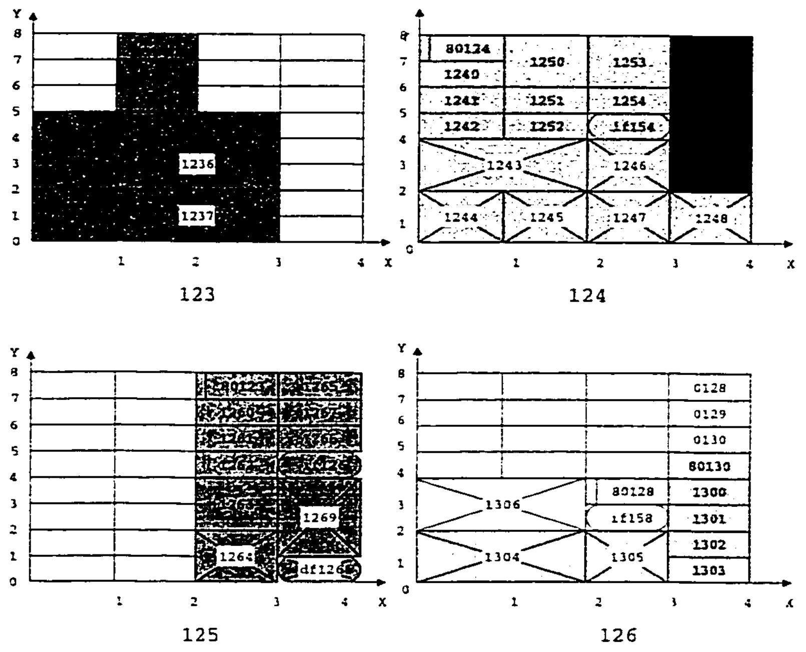


Figure B.2: Good examples of rule:NoLonelyDA - file: nolone.good (part 1)

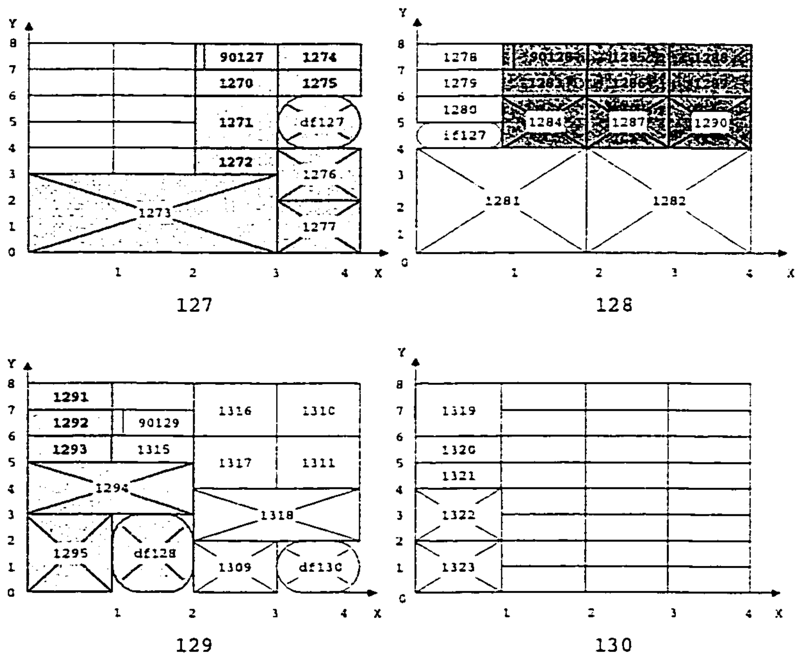


Figure B.3: Good examples of rule:NoLonelyDA - file: nolone.good (part 2)

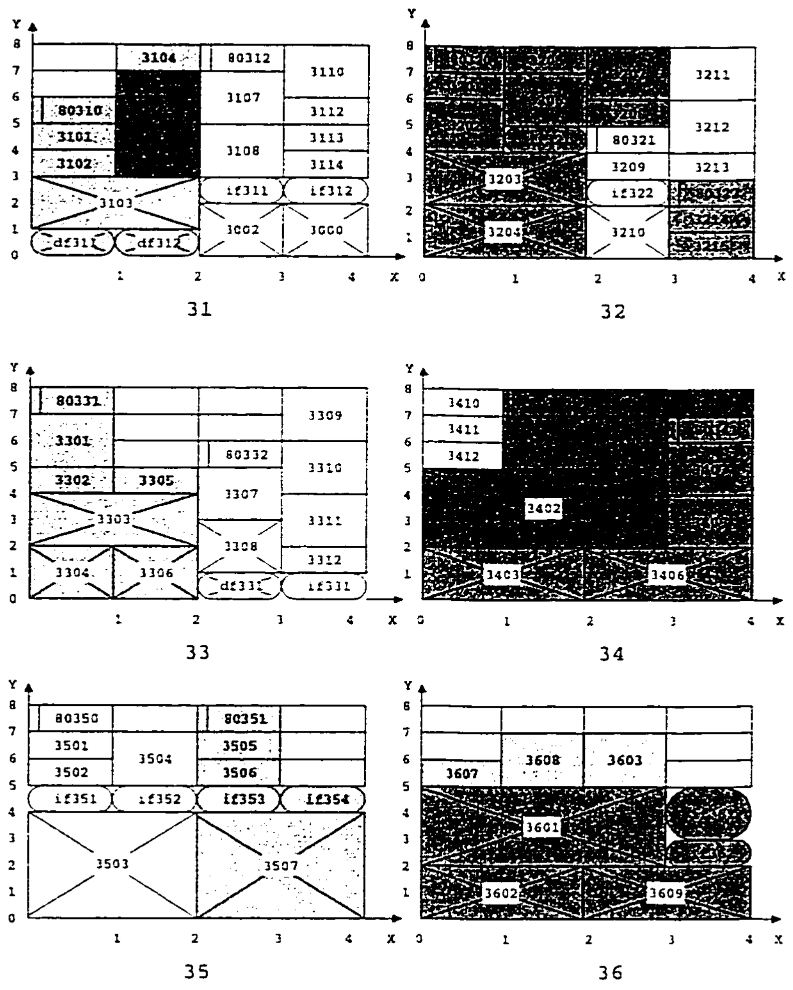


Figure B.4: Good examples of rule:NoIcUnderDA - file: da.good

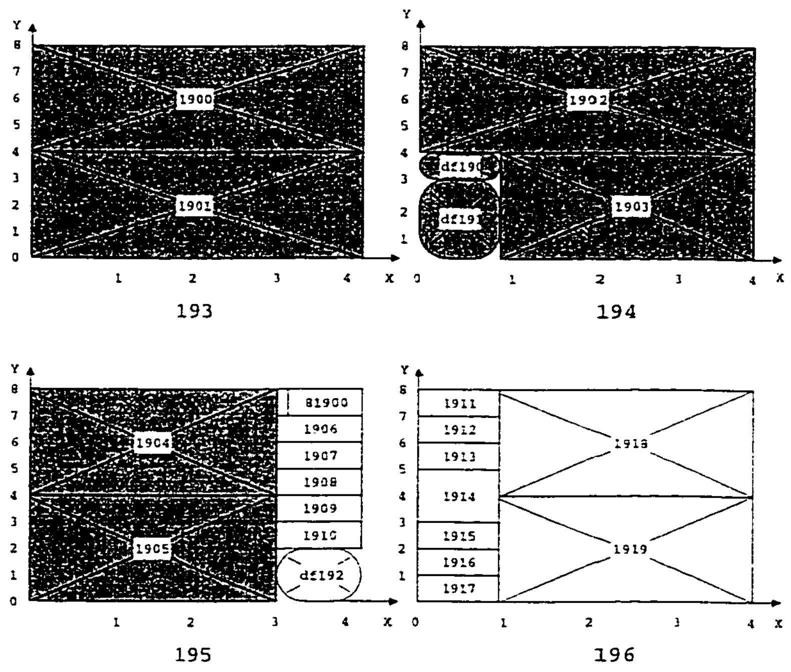


Figure B.5: Good examples of rule:NoICsAboveHPDA - file: hpda2.good (part 1)

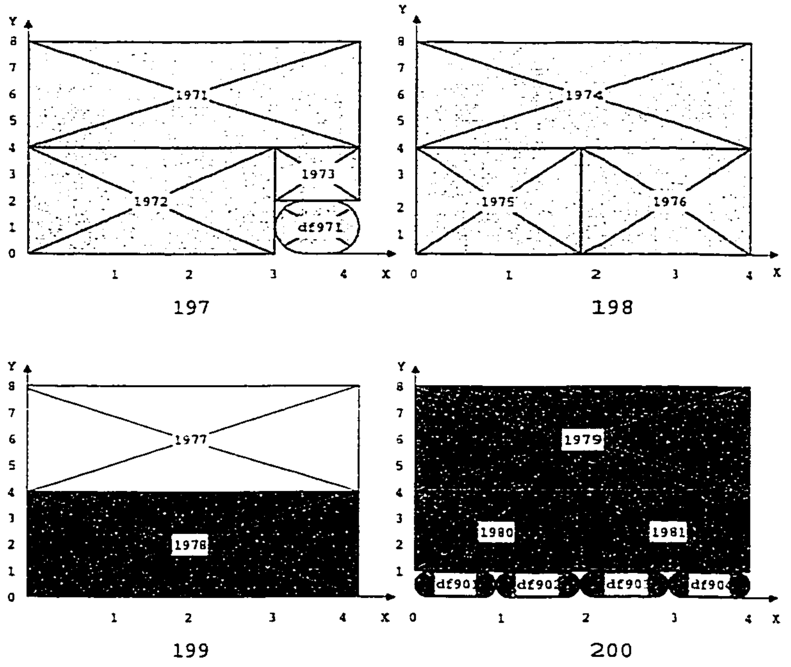


Figure B.6: Good examples of rule:NoICsAboveHPDA - file: hpda2.good (part 2)

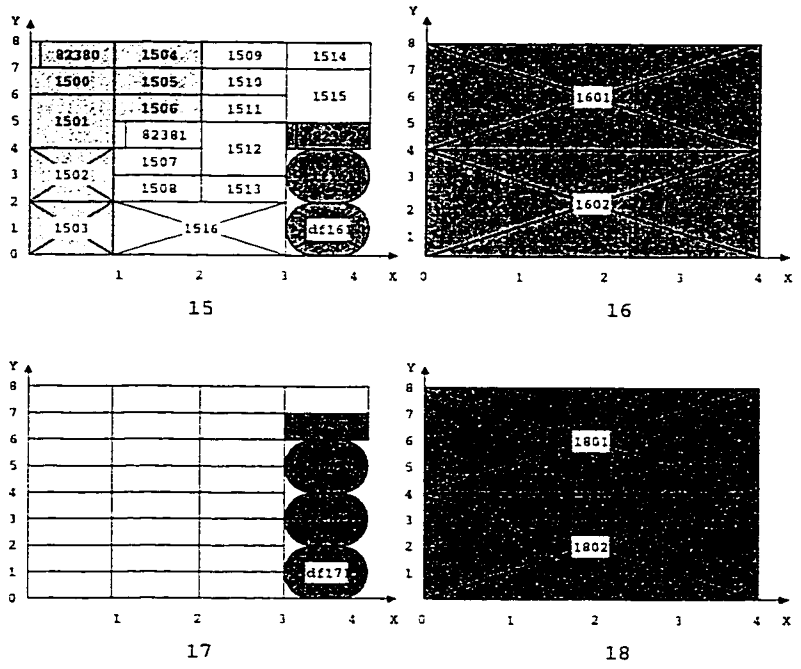


Figure B.7: Good examples of rule:NoICsBeforeHPDA - file: noicbf.good

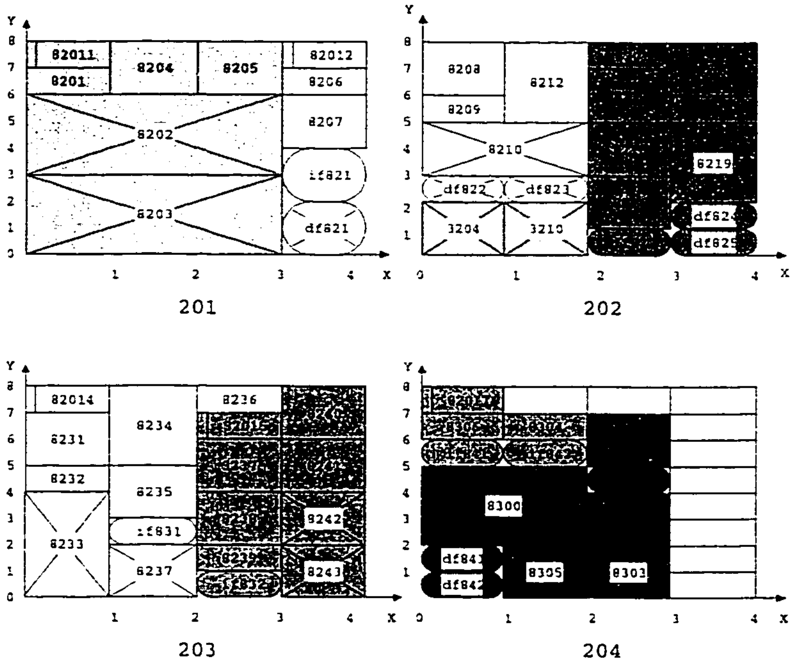


Figure B.8: Good examples of rule:NoIFsAtTopOfPage - file: if.good

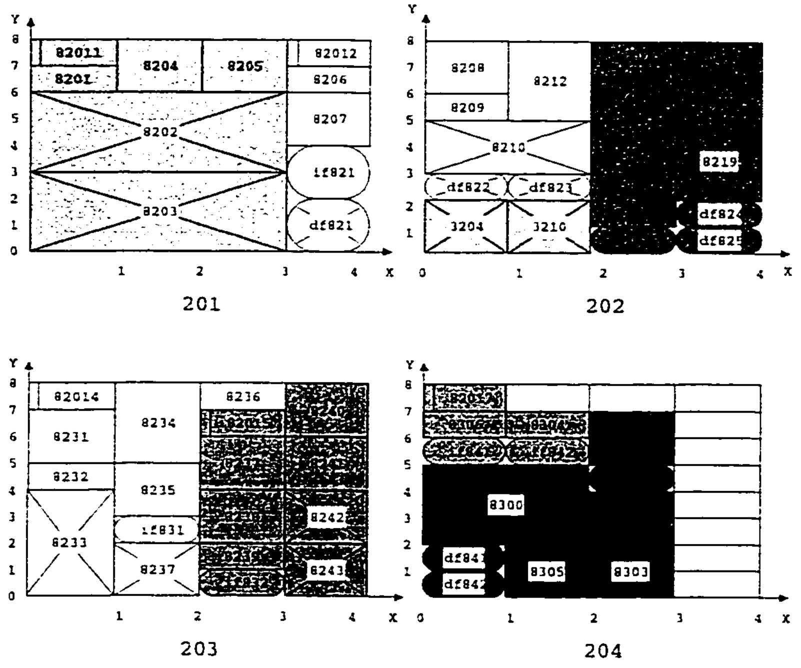


Figure B.9: Good examples of rule:DFsOnlyAtBottomOrUnderDA - file: df.good

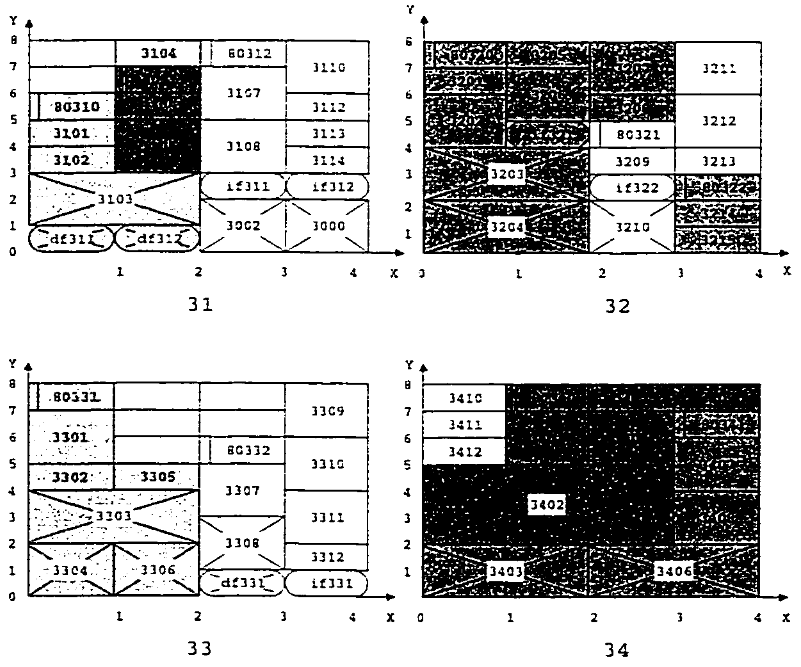


Figure B.10: Good examples of rule:Touching - file: touch.good

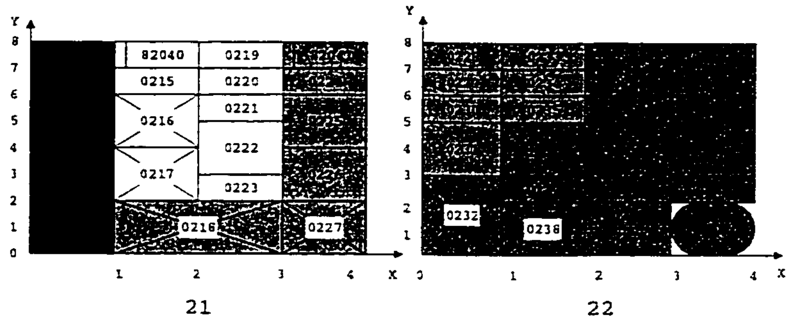


Figure B.11: Good examples of rule:RankByGridWithinHeading - file: rankgrid.good

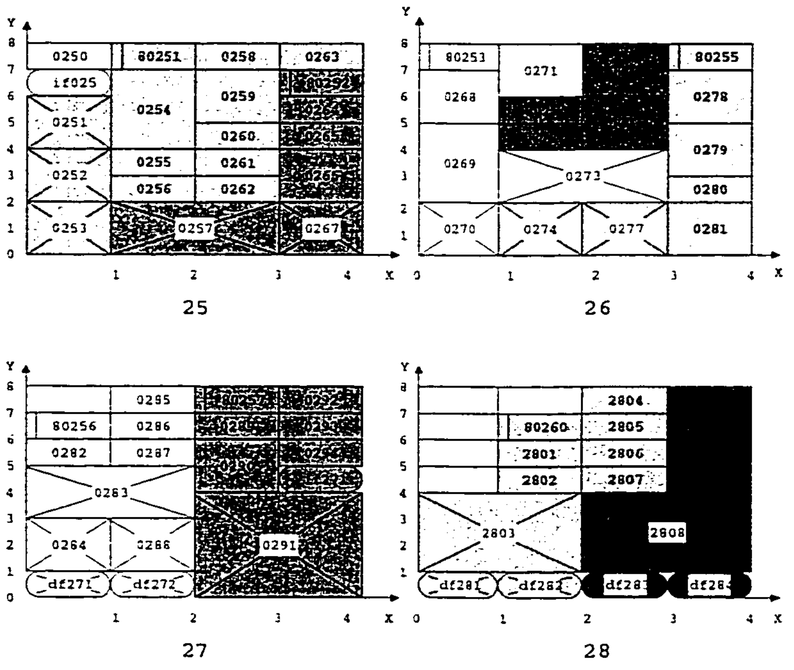


Figure B.12: Good examples of rule:ReadingOrder - file: rdorder.good

B.2 Bad Examples

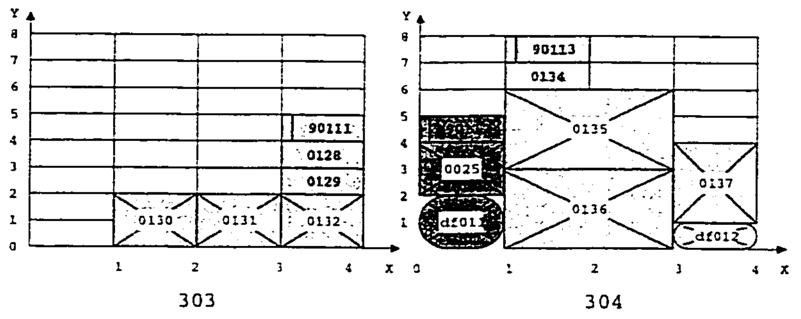


Figure B.13: Bad examples of rule:SmallDAsAndICsStartOnSamePage - file: smda.bad

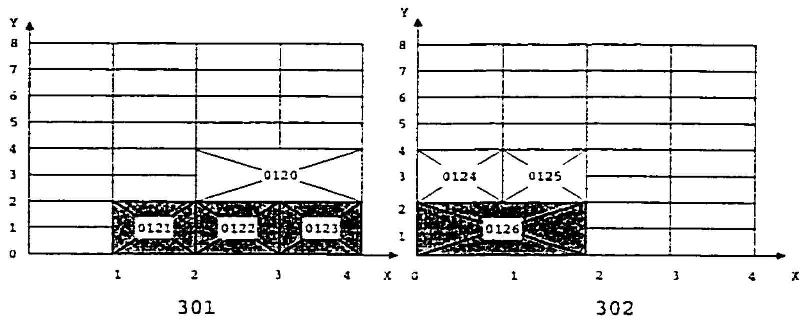


Figure B.14: Bad examples of rule:NoLonelyDA - file: nolone.bad

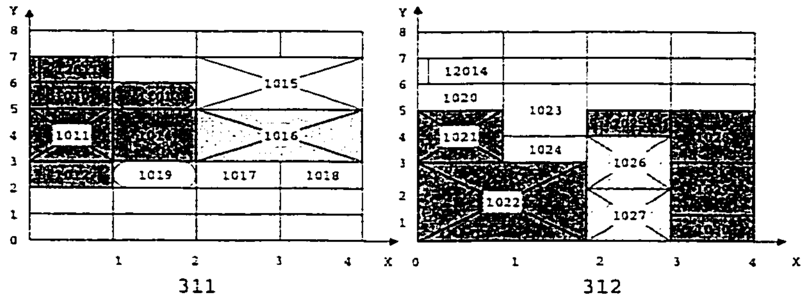


Figure B.15: Bad examples of rule:NoIcUnderDA - file: da.bad

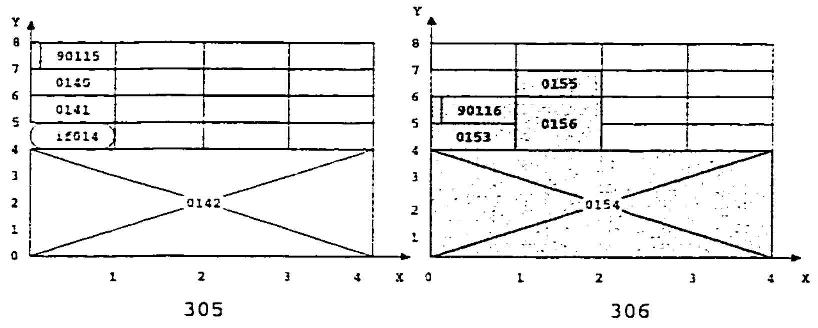


Figure B.16: Bad examples of rule:NoICsAboveHPDA - file: noicabv.bad

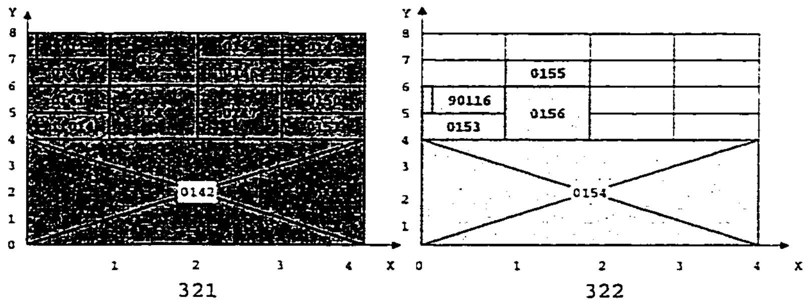


Figure B.17: Bad examples of rule:NoICsAboveHPDA - file: abv.bad

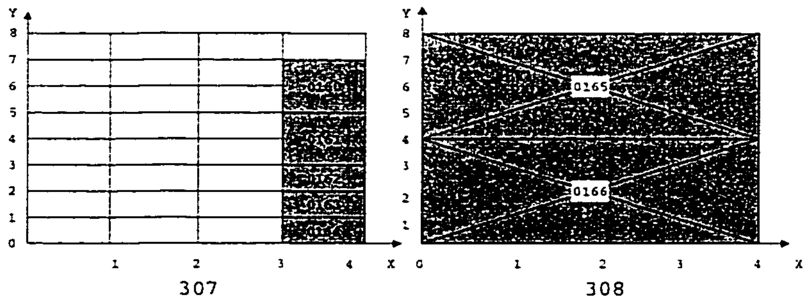


Figure B.18: Bad examples of rule:NoICsBeforeHPDA - file: noicbf.bad

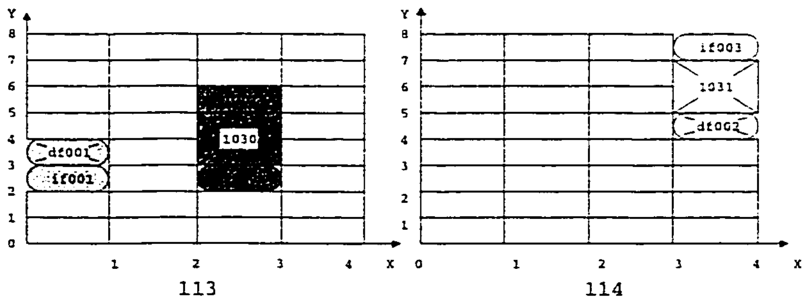


Figure B.19: Bad examples of rule:NoIFsAtTopOfPage - file: if.bad

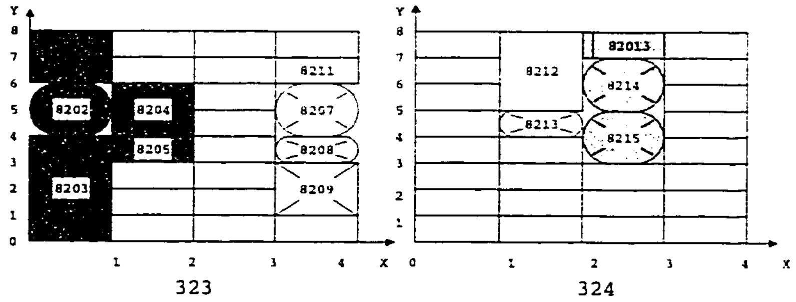


Figure B.20: Bad examples of rule:DFsOnyAtBottomOrUnderDA - file: dfl.bad

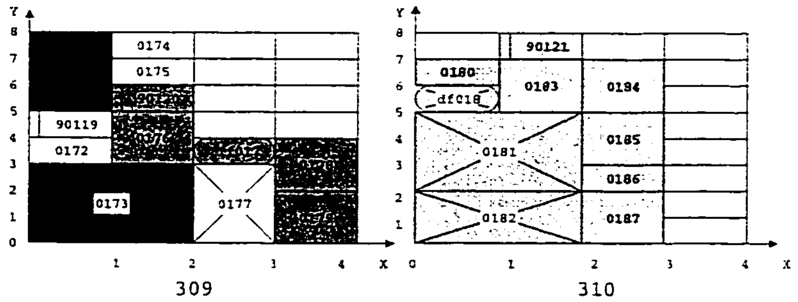


Figure B.21: Bad examples of rule:Touching - file: touch.bad

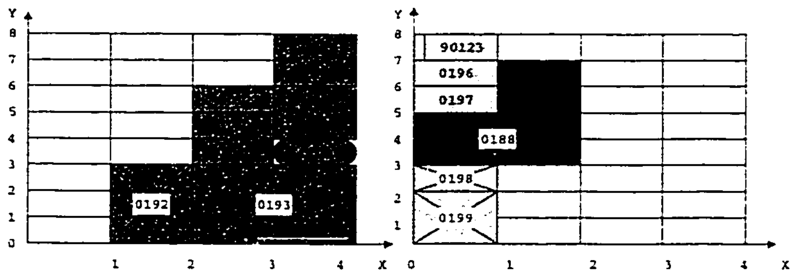


Figure B.22: Bad examples of rule:RankByGridWithinHeading - file: rank1.bad

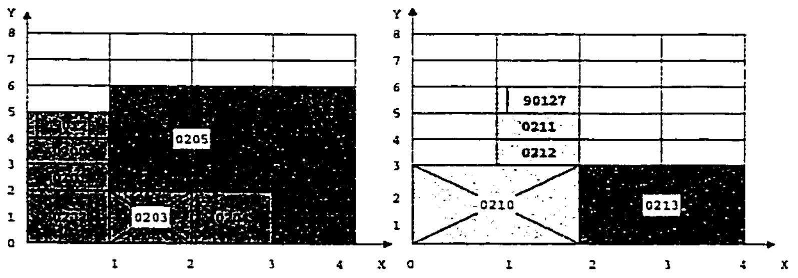


Figure B.23: Bad examples of rule:ReadingOrder - file: read.bad

Appendix C

Data Sets

The following three tables list each source filename and the corresponding file for Progol in task 1, task2 and task 3, respectively.

Good Example File	Bad Example File	Output File for Progol
smda1.good	smda.bad	smda.pl
nolone.good	nolone.bad	nolone.pl
da.good	da.bad	da.pl
da.good	da.bad	und.pl
noicbf.good	noicbf.bad	nicbf.pl
hpda2.good	abv.bad	hpda.pl
hpda2.good	noicabv.bad	above.pl
if.good	if.bad	if.pl
df.good	df1.bad	df1.pl
touch.good	touch.bad	touch.pl
rdorder.good	read.bad	read.pl
rankgrid.good	rank1.bad	rank.pl

Table C.1: Task 1: Input example files and their output

Good Example File	Bad Example File	Output File for Progol
da.good	da.bad	da2.pl
df.good	df.bad	df2.pl
if.good	if.bad	if2.pl
rdorder.good	ic.bad	ic2.pl
hpda2.good	hpda.bad	hpda2.pl

Table C.2: Task 2: Input example files and their output

Good Example File	Bad Example File	Output File for Progol
task3.good	t-da.bad	t-da.pl
task3.good	t-df.bad	t-df.pl
task3.good	t-if.bad	t-if.pl
task3.good	t-ic.bad	t-ic.pl
thpda.good	t-hpda.bad	t-hpda.pl

Table C.3: Task 3: Input example files and their output