

# LEARNING SEMANTIC MODELS AND GRAMMAR RULES OF BUILDING PARTS

Y. Dehbi, J. Schmittwilken, L. Plümer

Institute of Geodesy and Geoinformation, University of Bonn, Germany  
{dehbi, schmittwilken, pluemer}@igg.uni-bonn.de

**KEY WORDS:** Machine Learning, Inductive Logic Programming, Progol, Attribute Grammar, 3D Model, Semantic Model

## ABSTRACT:

Building reconstruction and building model generation nowadays receives more and more attention. In this context models such as formal grammars play a major role in 3D geometric modelling. Up to now, models have been designed manually by experts such as architects. Hence, this paper describes an Inductive Logic Programming (ILP) based approach for learning semantic models and grammar rules of buildings and their parts. Due to their complex structure and their important role as link between the building and its outside, straight stairs are presented as an example. ILP is introduced and applied as machine learning method. The learning process is explained and the learned models and results are presented.

## 1. INTRODUCTION

### 1.1 Motivation and Context

The last decade has seen an increasing demand for 3D building models. These models represent the basis of many approaches for the reconstruction and the generation of 3D urban scenes. For the interoperable access of 3D city models, the data model *CityGML* has been standardised (Kolbe and Groeger, 2006).

Apart from their original application in natural language processing, formal grammars have been used for a variety of approaches in 3D modelling, e.g. generation of synthetic city models and also geometrical reconstruction of urban objects such as buildings and building parts. Figure 1 illustrates building models of the Wilhelminian quarter ‘Südstadt’ in Bonn, Germany, which were generated with formal grammars (Krückhans, 2009).



Figure 1. Wilhelminian style building model in Bonn, generated with formal grammars

As yet grammar rules have to be defined by experts. An automatic generation of these rules has not been available until

now. In this paper we present a machine learning approach based on Inductive Logic Programming (ILP) with the aim of learning grammar rules. In order to express the constraints between the primitives of the modelled 3D objects, attributes and semantic rules are included. Through this approach the consistency and correctness of the 3D models is preserved. ILP has already been applied successfully to learn natural language grammar (Mooney, 1997). However, the learning of semantic models and grammar rules of building parts with ILP is a novel approach. The learning approach is demonstrated through an important and complex example of a building part, viz. straight stairs.

This paper is structured as follows: the next section reviews related works in the field of formal grammars and inductive logic programming. Section 3 introduces formal grammar and discusses the theoretical background of ILP as machine learning method. Section 4 presents the research results of our work, and finally, section 5 summaries the paper and gives an outlook.

## 2. RELATED WORK

Formal grammars have widely been applied in reconstruction tasks. Han (2009) and Zhu (2006) interpret single images using attribute graph-grammars. Their approach capitalises on grammars in a bidirectional way: The top-down derivation of the grammar generates model hypotheses which have to be verified subsequently. The bottom-up run aggregates the features to composite objects.

Mueller (2006, 2007) propose the application of context-sensitive split grammars which were introduced by Wonka (2003). The grammar controls the procedural modelling and synthetic generation of consistent mass models. The split rules ensure a tessellation of the faces in order to have topologically correct models.

Ripperda and Brenner (2009) and Ripperda (2008) use grammars for the reconstruction of windows from terrestrial facade images. Therefore they defined a grammar that describes

the general structure of facades, i.e. the alignment of windows in grids, both regular and irregular. The grammar controls the sampling of a reversible jump Markov Chain Monte Carlo process. Similar to Han (2009) they work on single images.

Schmittwilken (2009) presents an approach for the reconstruction of composite objects from 3D point clouds using attribute grammars. The composition is specified by attributes which define and propagate form parameters and location parameters of the modelled objects.

Apart from related work in reconstruction context, our approach draws upon ideas of ILP, which has been successfully applied as a learning approach in many different fields. In bioinformatics, for instance, ILP has been used to predict protein structure (Muggelton et al., 1992) and mutagenicity (King et al., 1996). Mooney (1997) has applied ILP to natural language processing.

### 3. BACKGROUND

In the following section, we will give an introduction to formal grammars and an overview of the applied machine learning approach ILP.

#### 3.1 Formal Grammars

Ever since formal grammars were introduced by Chomsky (1956, 1959) for reconstructing sentences of natural language, they have also been used to generate formal languages. A formal grammar  $G$  can be defined as quadruple  $\{S, N, T, P\}$  of a start symbol  $S$ , a set of non-terminals  $N$  represented by capitalised initials, a set of terminals  $T$  denoted by lower case initials and a set of production rules  $P$ .

A special case of formal grammars are context free grammars which play a major role in describing and designing most of the programming languages. Production rules appear in the form  $A \rightarrow a$  where  $A$  is a non-terminal, and  $a$  is a sequence of terminals and non-terminals. This rule implies that each occurrence of the symbol  $A$  can be replaced by the string  $a$ .

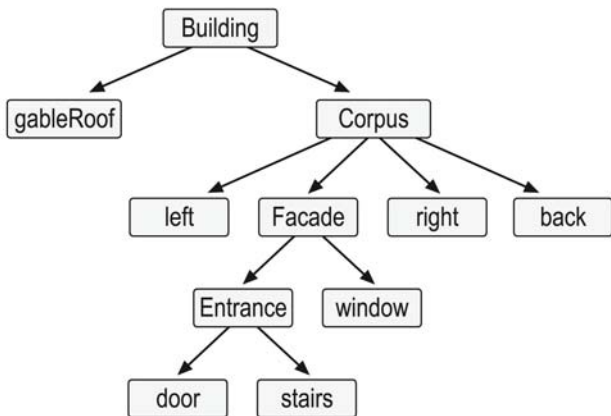


Figure 2. Derivation tree for the grammar rules of a building

A gable roof building can be modelled context free as follows: *Building* as a start symbol is made of *Corpus* and *gableRoof*. In addition to the *Facade*, *Corpus* consists of *left* side, *right* side,

and *back* side. *Facade* is built of *windows* and *Entrance* which is made of *doors* and *stairs*. This aggregation can be expressed by the production rules that are illustrated in the derivation tree in figure 2.

Likewise, *Building* can be represented by the following production rules  $P$ :

$P = \{$ $Building \rightarrow Corpus\ gableRoof$ $Corpus \rightarrow left\ Facade\ right\ back$ $Facade \rightarrow Entrance\ window$ $Entrance \rightarrow door\ stairs\}$
--

The set of the non-terminals is  $N = \{Building, Corpus, Facade, Entrance\}$ ; the set of terminals is  $T = \{gableRoof, left, right, back, window, door, stairs\}$ .

Context free grammars are suitable to describe context free languages. In this case the non-terminals can be substituted regardless of the context in which they occur. However, some structures can only be produced with regard to their context. In the case of stairs, for example, context free grammars are not expressive enough to state that all steps of the same stairs have the same dimensions.

$P_1: Stairs^{n-1} \rightarrow Step\ Stairs^n$ $P_2: Stairs \rightarrow Step$ $P_3: Step \rightarrow riser\ tread$ $R_1(P_1): Stairs^n.numberOfSteps = Stairs^{n-1}.numberOfSteps + 1$ $R_2(P_2): Step.width = Stairs.width$ $R_3(P_2): Step.height = Stairs.height$ $R_4(P_2): Step.depth = Stairs.depth$ ...
---

Table 1. Excerpt of an attribute grammar for stairs

In order to deal with this problem context free grammars have been extended by Knuth's attribute grammars (1968, 1971), which have been further extended by the probabilistic concept of an attribute grammar as proposed by Abney (1997). In attribute grammars Terminals and non-terminals are expanded by attributes, whereas production rules are extended by semantic rules. The latter specify the constraints among the attributes. An extract of an attribute grammar of stairs is shown in Table 1.

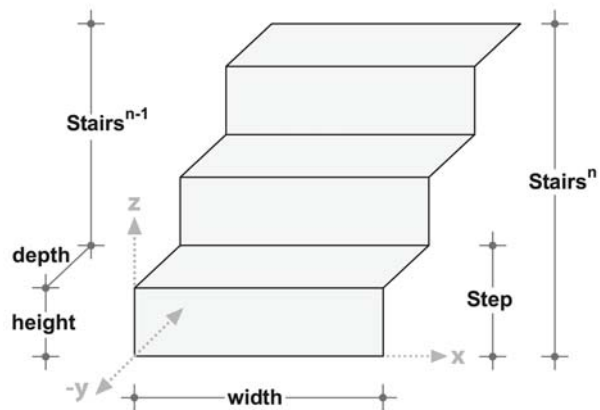


Figure 3. Stairs as recursion of steps

In contrast to the first example of the building grammar, stairs exhibit a recursive structure which allows for specifying an arbitrary number of steps. A staircase is shown in figure 3 by a sequence of steps which consist of risers (vertical rectangles) and treads (horizontal rectangles). The grammar symbols in table 1. are augmented by the attributes height, depth, and width, which are used in semantic rules  $R_2(P_2)$  to  $R_4(P_2)$  in order to specify the identity between the form parameters of risers and treads within the same stairs. The superscript indices  $n$  and  $n-1$  are used to differentiate between multiple occurrences of the same symbol.

### 3.2 Inductive Logic Programming

Inductive Logic Programming is a subarea of artificial intelligence which combines machine learning with logic programming. Thus, the goal of ILP, which is inherited from inductive machine learning, consists in developing techniques to induce hypotheses from observations as well as synthesising new knowledge from experience by using computational logic as representational schema.

In the following section some important concepts in logic programming are introduced (Raedt, 2008). First, a function is called *predicate*, if it returns a truth value, e.g. *parallel/2* is a predicate of arity 2. Second, a *term* is a constant, a variable or a structured term  $f(t_1, \dots, t_n)$  composed of a functor  $f$  and  $n$  terms  $t_i$ . Third, an *atom* is a predicate symbol followed by its necessary number of terms, e.g. *line(X)* is an atom which represents a line in a two dimensional space and which is represented by the term  $X$  as variable. Fourth, a *literal* is an atom or its negation.

By using these concepts the key concept of a *horn clause* can be defined as an expression of the form:  $h \leftarrow b_1, \dots, b_m$  in which  $h$  and  $b_i$  are logical atoms. The symbol ‘,’ symbolises a conjunction whereas ‘ $\leftarrow$ ’ stands for an implication. Clauses of the form:  $h \leftarrow true$  are called *facts*.

An example of a horn clause  $C$  can be illustrated by the *parallel* relation (here given in the logic programming language *Prolog*):

$$C: \text{parallel}(X,Y) \leftarrow \text{line}(X), \text{line}(Y), \text{line}(Z), \text{orthogonal}(X,Z), \text{orthogonal}(Y,Z).$$

Hereby a new predicate *parallel* is defined as *head* of the rule (the arrow’s right hand side being the rule’s *body*). The line  $X$  and the line  $Y$  satisfy this predicate if there is a line  $Z$  which is orthogonal to both lines  $X$  and  $Y$ . Moreover, a first order alphabet can be defined as a set of predicate symbols, constant symbols and functor symbols.

ILP systems are able to learn first order horn clauses by the use of background knowledge. Apart from syntactic differences, logic programs and attribute grammar are basically the same (Abramson and Dahl, 1989). Thus, the learned logic programs can be safely used for reconstruction tasks realised by attribute grammars.

The observation of 3D point clouds in order to recognise and extract constraints (cf. Schmittwilken, 2009) and the learning of 3D models from these constraints are two disjoint but complementary problems. Thus, this paper assumes that the user first provides examples such as sketches of the models which have to be learned. These are then processed either by

using snap points or thresholds to obtain consistent learning examples. This step is beyond the scope of this paper. Consequently, our focus and attention will be given to ILP as machine learning formalism.

Another opportunity to handle the uncertainty and to deal with noisy data consists in the extension of inductive logic programming techniques with probabilistic reasoning mechanisms. This area is called Probabilistic Inductive Logic Programming (Raedt et al., 2008).

#### 3.2.1 Method

Given a set  $E$  of positive and negative examples as ground facts and a background knowledge  $B$ , a hypothesis  $H$  has to be found which explains the given examples with respect to  $B$  and meets the language constraints. The hypothesis  $H$  has to be complete and consistent.  $H$  is complete if all the positive examples are covered. If none of the negative examples are covered  $H$  is considered consistent. The coverage of an example  $e \in E$  is tested with a function  $\text{covers}(B,H, e)$  which returns the value *true* if  $H$  covers  $e$  given the background knowledge  $B$ , and otherwise returns *false*.

The ability to provide declarative background knowledge to the learning system is one of the most distinct advantages of ILP. This background knowledge can be given in the form of horn clauses or ground facts which represent the prior knowledge about the predicates that appear in the learned hypothesis later. For example, if instances of the predicates *orthogonal* and *line* (cf. last section) are given, they can serve as background knowledge in order to learn the *parallel* relation.

ILP tasks are search problems in the space of possible hypotheses. In order to perform this search adequately a partial ordering of clauses is needed. For this purpose many inductive logic programming systems use  $\theta$ -subsumption as generalisation or specialisation operator. A clause  $C$  is said to be  $\theta$ -subsumed by a clause  $C'$  if there is a substitution  $\theta$ , such that  $C''\theta \subseteq C$ . Hereby a substitution  $\theta$  is a function which maps variables to terms. For instance, the clause:

$$C': \text{parallel}(X,Y) \leftarrow \text{line}(X), \text{line}(Y).$$

$\theta$ -subsumes the clause  $C$  in the last section under the empty substitution  $\theta = \emptyset$ . The  $\theta$ -subsumption defines the notion of generality. A clause  $C'$  is at least as general as clause  $C$  if  $C'$   $\theta$ -subsumes  $C$  (Lavrac, 1994).

#### 3.2.2 Aleph Algorithm

In the last few years many ILP systems have been implemented. One of the most popular ILP approaches is the *Progol* algorithm with many different implementations. Relevant for this paper is the *Aleph* engine (cf. Srinivasan, 2007) as a *Prolog* based implementation of *Progol*. *Progol* is an ILP framework which was developed to learn first order horn clauses.

For this aim, *Progol* first selects a positive seed example and then finds a consistent clause which is the most specific clause (MSC) of this example and which entails this example. Against the theoretical background of inverse entailment the MSC can be acquired (cf. Muggelton, 1995). The construction of the MSC will be exemplified in section 4.

In this way *Progol* learns by using a single example and by verifying the consistency of its generalisation with the dataset. This generalisation is added to the background knowledge. Afterwards, all redundant examples which have been covered by the MSC are removed. This process is repeated until a theory is found which covers all the positive examples. The coverage function is defined as follows:

$$\text{covers}(B,H,e) = \text{true if } B \cup H \models e$$

In other words, the hypothesis  $H$  covers the example  $e$  with respect to the background knowledge  $B$  if  $B \cup H$  semantically entails  $e$ .

In order to examine the goodness of clauses, each clause is evaluated by a score function. In this context, the default evaluation of *Aleph* is *coverage* which defines clause utility as  $P-N$ , in which  $P$ ,  $N$  are the number of positive and negative examples covered by the considered clause respectively. This could also be realised by other evaluation functions, like, for example, *entropy*.

As mentioned before, ILP tasks are search problems in the space of possible hypotheses. *Progol* bounds this space with the MSC as lower bound, whereas the top of the search space is bounded by the empty clause. Once the MSC has been built, *Aleph* performs a top-down search in the space of possible specialisations by using  $\theta$ -subsumption as refinement operator. In this process only the literals appearing in the MSC are used to refine clauses.

In order to restrict the vast space of hypotheses, learning systems employ a so-called declarative bias. ‘‘Bias’’ is anything other than the training instances that influences the results of the learner (Raedt, 2008). *Aleph* makes use of mode declarations as bias while searching for the best hypothesis. On the one hand, the mode declarations specify the predicates to be used, and state whether a predicate is a head (*modeh* declaration) or a body (*modeb* declaration) predicate. On the other hand, they declare whether an argument of a predicate may be an input variable, output variable or a constant value.

A further restriction of the space of the possible literals is achieved by type specification. Hereby a place-marker is employed in order to constrain the type of the terms that will replace the place-marker correctly. *Aleph* treats types just as labels and does not perform a type-checking. The different instantiations of a predicate in a given clause with the same sequence of input variables can also be bound through an integer, so-called *recall number*. An asterisk denotes that the predicate has unbounded indeterminacy. In *Aleph*, it is possible to infer mode and type information from the predicates of generative background knowledge and examples.

The declaration:

$$\text{modeb}(*, \text{orthogonal}(+line, -line))$$

states that the predicate *orthogonal* has to be used as body predicate and its arguments have to be instances of the same type. Additionally, the symbol ‘‘+’’ states that the first argument is an input argument and therefore it has to be bound, whereas the second is an output argument and may be either bound or unbound. In other words, in a literal *orthogonal*( $X$ ,  $Y$ ),  $X$  has to be an old variable, in contrast to  $Y$  that may be either an old or

a new variable. Within a clause any input variable in the body literal must be an input variable in the head of the clause or an output variable in some preceding literals.

#### 4. RESULTS

The following section presents the findings of our approach as well as the problems that have occurred in the process of learning. The potential to model background knowledge enables a stepwise learning process of stairs. In order to evaluate the potential of ILP for the automatic learning of building parts straight stairs have been used as an example. The specific challenge in the case of stairs is their recursive structure. At the beginning it was not clear whether ILP would meet this challenge. Interestingly, some positive and some negative examples were sufficient.

Thus, the major difficulty in learning how stairs are defined consists in handling the recursion. Due to the complexity of stairs, in particular their recursive structure, the learner requires a large amount of negative examples. However, the learning process becomes much easier by applying a modular bottom-up approach, which consists of two steps: First, the non-recursive parts of stairs are learned, and then, building up on these parts, the recursive clause is learned. This divide and conquer approach is illustrated in figure 4. Stairs are broken down into their primitives. Aggregation and learning directions are indicated by descending and ascending arrows respectively. Aggregation occurs top-down whereas learning is realised by a bottom-up approach, starting from the smallest atomic feature to the whole *stairs* object. The semantic primitives are marked in bold whereas the topologic primitives are marked in normal font.

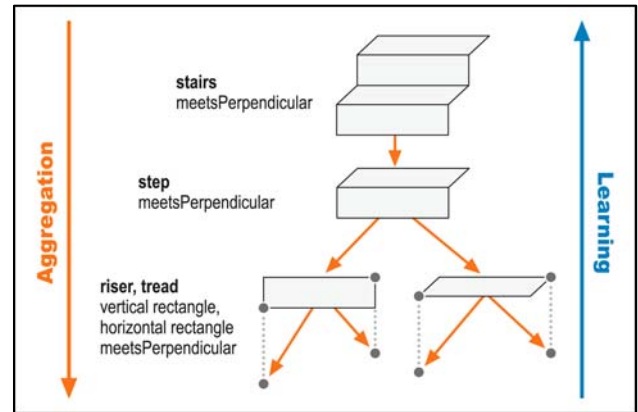


Figure 4. Semantic and topologic primitives of stairs

Table 2 gives the produced output rules of the learner. In order to learn the end recursive rule in lines (01-05) only four positive and two negative examples were required. This low number can be attributed to the stepwise learning strategy. The interested reader may verify that the difference between these rules and those in table 1 is merely syntactic. It should be noted that rules in table 1 are only a subset of the rules of table 2.

On the level of aggregation we have seen in section 3.1 that stairs are made of a recursion of steps that are composed of horizontal and vertical rectangles. In turn both rectangles can be defined by two 3D points. In contrast to the aggregation, the starting point on the level of learning is two observed left and

```

01 stairs([step(R,T,Point,Height,Depth,Width)], Height,Depth,Width) .
02 stairs([step(R2,T2,Point2,Height,Depth,Width),
03         step(R1,T1,Point1,Height,Depth,Width) | Tail], Height,Depth,Width)
04     ← stairs([step(R1,T1,Point1, Height,Depth,Width) | Tail], Height,Depth,Width) ,
05         meetsPerpendicular(T1,R2) .
06
07 step(R,T,Point1,Height,Depth,Width) ← riser(R,Point1,Height,Width) ,
08         tread(T,Point2,Depth,Width) ,
09         meetsPerpendicular(R,T) .
10
11 meetsPerpendicular(R,T) ← riser(R,Point(X,Y,Z),Height,Width) , plus(Z,Height,Z1) ,
12         tread(T,Point(X,Y,Z1),Depth,Width) .
13
14 riser(R,point(X,Y,Z),Height,Width) ← plus(X,Width,X1) , point(X1,Y,Z1) , plus(Z,Height,Z1) .
15
16 tread(T,point(X,Y,Z),Depth,Width) ← plus(X,Width,X1) , point(X1,Y1,Z) , plus(Y,Depth,Y1) .
17
18 meetsPerpendicular(T,R) ← tread(T,point(X,Y,Z),Depth,Width) , plus(Y,Depth,Y1) ,
        riser(R,point(X,Y1,Z),Height,Width) .

```

Table 2. The whole learned logic program of stairs

right 3D points for identifying the horizontal (tread) and the vertical rectangle (riser) respectively. As described in subsection 3.2.1, the task of learning requires a set of positive examples which are generalised with respect to the set of negative examples and the background knowledge. *Aleph* supports first order horn clauses as background knowledge and is further able to learn ranges and functions with numeric data. These functions can also be used as background knowledge that represents a good basis to describe the geometric and topologic constraints inside building parts. This is particularly important at the low level of the learning process. Beside the observed point in the case of *riser*, the background knowledge includes arithmetic, namely the operation *plus* which is used to specify topologic constraints between these points of the rectangle and their coplanarity. Since straight stairs are invariant in rotation we can safely assume an axis-parallelity with regard to the representation in figure 3.

Altogether two positive examples and two negative examples were necessary to learn a *riser*. The learning of the concept of *tread* happened analogously. In order to learn a *tread* the same number of examples was required. Rules in lines 14 and 16 (cf. table 2) show the learned rules of *riser* and *tread*.

Once *riser* and *tread* have been learned, they can be used as primitives in order to learn the concept of a step. They are added to the background knowledge. In the following the learning process will be exemplarily elaborated.

For the completion of the necessary background knowledge of step, information about the adjacency between risers and treads is required. This is expressed by the relation *meetsPerpendicular* which will be explained later. Now we are able to learn the concept of step.

The rule in lines (07-09) (cf. table 2) shows the result clause, which defines a step. The head of this rule represents the whole step object, whereas the body defines its aggregated primitives. The attributes *R* and *T* serve as an identifier for *riser* and *tread* respectively which the whole step is composed of. The geometric description of the model is given by the location and form parameters. The location parameter of step is described by the attribute *Point1*. Figure 5

demonstrates that this attribute represents the left point of its riser. The remaining attributes *Height*, *Depth* and *Width* constitute the form parameters. For the consistency of the model the locations as well as the form parameters have to be propagated, that is, they occur as attributes in the head of the rule.

The semantic rules are implicitly specified by the identity of the attributes and the relation *meetsPerpendicular*. On the one hand, the identity of the attribute *Width* in *riser* and *tread* represents the constraint that within the same step the riser and the tread have the same width. On the other hand, the occurrence of this attribute in the head ensures its propagation from the parts to the whole step. In contrast, the values of *Height* and *Depth* are only inherited as property of *step* without being further constrained. In addition to the identifier *R* and *T* both the location parameter *Point1* and the form parameters *Height*, *Depth* and *Width* are propagated. We have already explained that the symbol ‘;’ in the rule must be interpreted as a logical AND. This rule means that *step* is a valid step if a *riser* and a *tread* exist which have the same *Width* and which are further perpendicularly connected.

As mentioned in section 3.2.2 we demonstrate the construction of the MSC using an example of step. In order to form a rule for a step, *Aleph* searches the subset of the background knowledge which is relevant for a seed example and conforms to the mode declarations defined by the user. Table 3. shows an instantiation of these declarations for the case of step. The target predicate is declared with the *modeh* declaration which also specifies the necessary attributes of a step. The other declarations represent the possible predicates that may appear in the body of the learned rule.

```

modeh(*,step(+riser,+tread,+point,+height,
            +depth,+width)).
modeb(*,riser(+riser,+point,+height,+width)).
modeb(*,meetsPerpendicular(+riser,+tread)).
modeb(*,tread(+tread,-point,+depth,+width)).

```

Table 3. Mode declarations of step

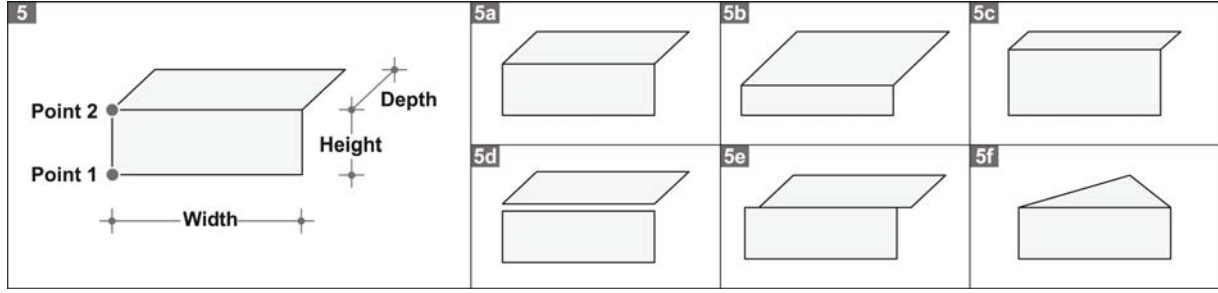


Figure 5: Parameters of a step; 5a, 5b, 5c: Positive examples of a step; 5d, 5e, 5f: Negative examples of a step

At the beginning of the learning process, *Aleph* picks up a seed example such as  $\text{step}(r, t, p, 1, 2, 3)$  provided by the user. This fact describes that a step is composed of riser  $r$  and tread  $t$  with  $\text{Height} = 1$ ,  $\text{Depth} = 2$  and  $\text{Width} = 3$  respectively. We assume that the following facts are available in the background knowledge:  $\text{riser}(r, p, 1, 3)$ ,  $\text{tread}(t, p', 2, 3)$  and  $\text{meetsPerpendicular}(r, t)$ . In this case the identity of the constants  $r$  and 3 in  $\text{step}$  and  $\text{riser}$  is decisive to specify how relevant the fact  $\text{riser}(r, p, 1, 3)$  is for  $\text{step}(r, t, 1, 2, 3)$ . In the same manner the remaining facts will be considered as relevant. Consequently the following rule is formed from these facts:

$$\text{step}(r, t, p, 1, 2, 3) \leftarrow \text{riser}(r, p, 1, 3), \text{tread}(t, p', 2, 3), \text{meetsPerpendicular}(r, t).$$

Now this rule is generalised to the following clause which meets the mode declarations presented in table 3.:

$$\text{step}(R, T, P, \text{Height}, \text{Depth}, \text{Width}) \leftarrow \text{riser}(R, P, \text{Height}, \text{Width}), \text{tread}(T, P', \text{Depth}, \text{Width}), \text{meetsPerpendicular}(R, T).$$

The relation  $\text{meetsPerpendicular}$  in Table 2 ensures the topological correctness of the step model. A 3D model is said to be topologically correct if only  $\text{meets}$  and  $\text{disjoint}$  are allowed as spatial relations between its different objects (Egenhofer and Franzosa, 1991), viz, that geometric objects do not intersect or cover one another. This is assured by the arithmetic operation  $\text{plus}$  which moves the left point of  $\text{riser}$  by the distance  $\text{Height}$  along the  $z$ -axis. The perpendicularity is ensured by the vertical and horizontal alignment of riser and tread respectively.

In order to understand how the examples influence the learning, an excerpt of positive and negative examples which has been used to learn a step is illustrated in figure 5. The first three examples are positive examples, whereas the last three are negative examples. The examples 5a, 5b and 5c represent correct steps in which riser and tread have various  $\text{Depth}$  and  $\text{Height}$  and are perpendicular connected to build a step with a mutual edge. The example 5d avoids that a tread and a riser are validated to a step although they do not meet. Example 5e cannot represent a step because the constraint enforcing the movement of the left point of the riser along the  $z$ -axis is not satisfied. The last example 5f illustrates the case that the riser meets perpendicularly with another geometric object which does not correspond to a tread. All in all three positive and three negative examples were needed to learn the rule of step.

The concept of steps is only one part of the concept of stairs. The adjacency between steps has to be learned as well. In contrast to the relation  $\text{meetsPerpendicular}$  in line 11

which ensures the topologic correctness within a step the relation  $\text{meetsPerpendicular}$  in line 18 ensures this correctness between two neighbouring steps within stairs.

Now we have the necessary background knowledge to learn the recursive concept of stairs. The rule for stairs has already been shown in table 2 (cf. lines 01-05). The square brackets in this rule symbolise a list in *Prolog*. In other words, stairs are represented as a list of steps which is separated into head and tail ( $\text{stairs}([\text{head}|\text{tail}])$ ). The new stairs on the left side of the rule consists of the stairs from the right side concatenated with the next new neighbouring step. As mentioned above, the neighbourhood of the steps is ensured with the  $\text{meetsPerpendicular}$  relation. This concatenation of steps implies that the new stairs consists of exactly one step more than the one before. Once again it should be noted that four positive examples and two negative examples were sufficient to learn the recursive clause  $\text{stairs}$ .

*Aleph* is one of the so-called batch learners. This means that the examples which are labelled positive or negative have to be provided before the learning process takes place, in contrast to incremental learners which expect that the examples will be given step by step. In this context the positive and the negative examples of the concept to be learned are written in two separate files. The background knowledge is delivered additionally in a third file which contains clauses and facts that represent logically encoded information relevant to the domain of interest.

Furthermore, the validity of the generalised models has to be verified by a user. Thus, *Aleph* belongs to the category of interactive systems. Since attribute grammar rules are generative, the user is able to test and adjust the temporary results and classify new examples until a valid model of the target concept has been generated.

## 5. SUMMARY AND OUTLOOK

We have introduced an ILP based approach for learning semantic models and grammar rules of building parts and have taken straight stairs as an example to demonstrate our machine learning approach. Beside the geometric and the topological constraints, the specific challenge lies in handling the complex recursive structure. This complexity has been overcome by using a bottom-up approach consisting in learning non-recursive primitives of stairs first, and then learning the whole stairs.

The ability to incorporate background knowledge turns out to be a major tool to cope with this kind of complexity. The learned

model is correct and consistent from a geometrical, topological and semantic point of view. A very limited number of examples were sufficient to explain a human understanding of stairs to the machine.

Recursion did not turn out to be a major obstacle. Consequently, this is the first step of a major project for an interactive generating of semantic models and attribute grammar rules of man-made objects such as buildings.

This paper proposed a machine learning method for 3D models which requires examples provided by the user. Another possibility is to learn such models by taking into consideration observed noisy data. In order to deal with these uncertainties and deviations, our future works will include probabilistic logic models in the learning process.

## 6. REFERENCES

- Abney, S., 1997. Stochastic Attribute-Value Grammars. *Computational Linguistics*, 4, pp. 597 - 618
- Abramson, H., Dahl, V., 1989. *Logic grammars*. Springer, New York
- Chomsky, N., 1956. Three models for the description of language. *Information Theory, IEEE Transactions*, 3, pp. 113 - 124
- Chomsky, N., 1959. On Certain Formal Properties of Grammars. *Information and Control*, pp. 137-167
- Egenhofer, M., Franzosa, R.D., 1991. Point-Set Topological Spatial Relations. *International Journal of Geographical Information Systems*, 2, pp. 161-174
- Kolbe, T.H., Gröger, G., Czerwinski, A. 2006. *City Geography Markup Language (CityGML), Implementation Specification Version 0.3.0, Discussion Paper, OGC Doc. No. 06-057*. Open Geospatial Consortium
- Han, F., Zhu, S.-C., 2009. Bottom-Up/Top-Down Image Parsing with Attribute Grammar. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1, pp. 59-73
- King, R. D., Muggleton, S., Srinivasan, A., Sternberg, M. J. E. 1996. Representing molecular structure activity relationships: the use of atoms and their bond connectivities to predict mutagenicity using inductive logic programming. *Proc. Nat. Acad. Sci. USA*, 1993, 438-442.
- Knuth, D.E., 1968. Semantics of context-free languages. *Theory of Computing Systems*, 2, pp. 127 - 145
- Knuth, D.E., 1971. Top-down Syntax Analysis. *Acta Informatica*, 2, pp. 79-110
- Krückhans, M., Schmittwilken, J. 2009. Synthetische Verfeinerung von 3D-Stadtmodellen. In: (eds): *Proceedings of DGPF-Jahrestagung, March 2009, Jena, Germany (Translation: Synthetic Refinement of 3D City Models)*
- Lavrac, N., Dzeroski, S., 1994. *Inductive logic programming: Techniques and applications*. Horwood, New York
- Mitchell, T.M., 1997. *Machine Learning*. McGraw-Hill, New York
- Mooney, R.J. 1997. Inductive Logic Programming for Natural Language Processing. In: (eds): *In Muggleton, S. (Ed.), Inductive Logic Programming: Selected papers from the 6th International Workshop*. Springer-Verlag, pp 3 - 24
- Muggleton, S., King, R.D. , Sternberg, M.J.E. 1992. Protein secondary structure prediction using logic. In: Muggleton S (eds): *ILP92*, pp 228-259
- Muggleton, S., 1995. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, pp. 245 - 286
- Muggleton, S., Raedt, L. de, 1994. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, pp. 629 - 679
- Müller, P., Wonka, P., Haegler, S. , van Ulmer, A.a.G.L., 2006. Procedural Modeling of Buildings. *ACM Transactions on Graphics*, 3, pp. 614 - 623
- Müller, P., Zeng, G., Wonka, P. , van Gool, L. 2007. Image-based Procedural Modeling of Facades. In: (eds): *Proceedings of ACM SIGGRAPH 2007 / ACM Transactions on Graphics*. ACM Press, New York, NY, USA, p 85
- Raedt, L. De, 2008. *Logical and relational learning: With 10 tables*. Springer, Berlin
- Raedt, L. De, Frasconi, P., Kersting, K., Muggleton, S.H. (Eds.), 2008. *Probabilistic inductive logic programming: Theory and applications ; [result of European IST FET project no. FP6-508861 on application of probabilistic inductive logic programming (APRIL II, 2004-2007)]*. Springer, Berlin
- Srinivasan, A., 2007. The Aleph Manual Version 4 and above. [http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/a\\_aleph\\_toc.html](http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/a_aleph_toc.html) (accessed 30 October 2009)
- Ripperda, N., 2008. Grammar Based Facade Reconstruction using RjMCMC. *Photogrammetrie, Fernerkundung, Geoinformation*, pp. 83 - 92
- Ripperda, N., Brenner, C. 2009. Application of a Formal Grammar to Facade Reconstruction in Semiautomatic and Automatic Environments. In: (eds): *Proc. of 12th AGILE Conference on GIScience*
- Schmittwilken, J., Plümer, L. 2009. Model Selection for Composite Objects with Attribute Grammars. In: (eds): *Proc. of 12th AGILE Conference on GIScience*
- Wonka, P., Wimmer, M., Sillion, F. , Ribarsky, W., 2003. Instant Architecture. *ACM Transactions on Graphics*, 4, pp. 669-677
- Zhu, S.-C., Mumford, D., 2006. A Stochastic Grammar of Images. *Foundations and Trends in Computer Graphics and Vision*, pp. 259 - 362