Proceedings of the Twenty-Eighth AAAI Conference on Innovative Applications (IAAI-16)

Automated Capture and Execution of Manufacturability Rules Using Inductive Logic Programming

Abha Moitra
Senior Computer Scientist
Knowledge Discovery Lab
GE Global Research (GRC)
Niskayuna, NY, USA, 12309
moitraa@ge.com

Ravi Palla
Computer Scientist
Knowledge Discovery Lab
GE Global Research (GRC)
Niskayuna, NY, USA, 12309
palla@ge.com

Arvind Rangarajan
Mechanical Engineer
Model-Based Manufacturing Lab
GE Global Research (GRC)
Niskayuna, NY, USA, 12309
Arvind.Rangarajan@ge.com

Abstract

Capturing domain knowledge can be a time-consuming process that typically requires the collaboration of a Subject Matter Expert and a modeling expert to encode the knowledge. In a number of domains and applications, this situation is further exacerbated by the fact that the Subject Matter Expert may find it difficult to articulate the domain knowledge as a procedure or rules, but instead may find it easier to classify instance data. To facilitate this type of knowledge elicitation from Subject Matter Experts, we have developed a system that automatically generates formal and executable rules from provided labeled instance data. We do this by leveraging the techniques of Inductive Logic Programming (ILP) to generate Horn clause based rules to separate out positive and negative instance data. We illustrate our approach on a Design For Manufacturability (DFM) platform where the goal is to design products that are easy to manufacture by providing early manufacturability feedback. Specifically we show how our approach can be used to generate feature recognition rules from positive and negative instance data supplied by Subject Matter Experts. Our platform is interactive, provides visual feedback and is iterative. The feature identification rules generated can be inspected, manually refined and vetted.

Introduction

The ability to capture domain knowledge is a critical task in many domains and applications. Quite often knowledge capture poses a roadblock in quickly developing and deploying systems that automate processing or reasoning tasks. For instance, a Subject Matter Expert (SME) might have deep domain knowledge but may not be able to describe it in terms of concepts and relationships that can be used for representing the knowledge. Also at times, the SME may not be able to describe the knowledge at the right level of detail that may be needed for making automated decisions.

In order to overcome such issues, we use an Inductive Logic Programming (ILP) based approach wherein the SME essentially identifies positive and negative examples for describing a concept and the ILP system uses them to derive logic programming rules for formally defining the concepts. Our approach is iterative in the sense that the SME can refine the rules learned by adding more positive and negative examples.

Our approach relies on a semantic model (consisting of ontologies and rules) that initially describes the basic concepts and relationships of the domain. To learn definitions of more complex concepts, the SME provides positive and negative examples that are automatically translated into a formal representation using the basic concepts and relationships.

The complex concepts thus learned are added back to the semantic model and this process is repeated to learn multiple levels of knowledge.

Our approach addresses both the issues described above. Since we automatically translate the example data provided by SME into a logical representation, the SME is not required to have knowledge of the concepts and relationships in the ontology. Also, since the SME only identifies positive and negative examples and repeats the learning approach until the knowledge learned is satisfactory, it provides a way for deriving complete and accurate descriptions of the concepts in the domain.

The rest of the paper is organized as follows. We start by briefly introducing ILP and then we describe the manufacturing domain where we applied this approach. We then discuss the Integrated DFM Learning Platform that we have developed and deployed, and illustrate our entire approach with examples. We then highlight some results that

we obtained by applying the approach on the domain. Finally we discuss related work and conclusions.

Inductive Logic Programming

Inductive Logic Programming (Muggleton 1991) is a branch of Machine Learning that deals with learning theories in the form of logic programs.

Given background knowledge (B) in the form of a logic program, and positive and negative examples as conjunctions E^+ and E^- of positive and negative literals respectively, an ILP system derives a logic program H such that:

- all the examples in E^+ can be logically derived from B ^ H, and
- no negative example in E^- can be logically derived from B ^ H.

ILP has been successfully used in applications such as bioinformatics and Natural Language Processing (Bratko and Muggleton 1995), (Chen and Mooney 2011), (Faruquie, Srinivasan, and King 2013). A number of ILP implementations are available. Examples include Aleph[1], Progol[2] (Muggleton 1995), and Atom[3]. In our work, we use Aleph with SWI-Prolog[4] (Wielemaker et al. 2012).

Design For Manufacturability

Design for manufacturability (DFM), a term used for incorporating manufacturing feedback during design phase, has gained popularity among design engineers and original equipment manufacturers (OEMs). DFM has become an integral part of the product development process. The main goal of DFM systems is to decrease iterations between design and manufacturing, thereby resulting in reduced lead time for new product introduction.

A typical design rule consists of a geometric feature, parameters associated with feature, and constraints on the parameters that define the bounds of manufacturability. For example, a rule for sheet metal hole drilling states "hole diameter must be at least equal to the sheetmetal thickness" (Radhakrishnan et al., 1996). Any implementation of this rule must start with definition of the hole feature. The diameter is the parameter associated with the hole feature and the thickness constrains the parameter. So, a primary requirement for a DFM system is the ability to recognize features.

Automated Feature Recognition

Automated Feature Recognition (AFR) plays a critical role in DFM as manufacturability rules can be defined for features. Extensive work has been done on AFR and (Babic, Nesic, and Miljkovic 2008) provide a detailed survey. While there are a number of different approaches, we focus on rule-based approaches as the results can more easily be examined and refined by domain experts. So, for our purposes, features are defined in terms of geometric and topological information and can be encoded as if-then rules. These rules are typically composed of attributes / properties of faces, edges, vertices.

(Brousseau, Dimov, and Setchi 2008) propose a method to automatically generate feature recognition rules using an inductive learning algorithm on training data consisting of feature examples. While their approach seems very similar to our approach, the key difference is that the feature examples they provide are all "simple" (i.e. the entire part has just that one feature) and they consider only planar and cylindrical faces (though they say that their approach can be extended). Our approach works with detailed actual parts including compressor casings and so allows us to generate AFR rules that handle complexities found in actual parts. We have automatically generated rules for tapers, spot faces, pad fillets, flange faces etc. Our platform is also interactive so we can consider one feature at a time building on previously generated feature recognition rules and it is integrated with CAD tool for visualization.

Once feature recognition rules are developed, they can then be incorporated in a knowledge base (Phelan, Wilson, and Summers 2014). These knowledge bases can also be used in various ways including generating explanations for AFR (Wang 2012).

Commercial DFM Tools

There are several DFM tools currently available either as a standalone package or integrated with CAD tools. Some of them are DFMPro from Geometric[5] that is available in SolidWorks[6], Checkmate in NX[7], DFM tool from Boothroyd and DewHurst Inc.[8] and Apriori[9]. These tools consist of several rules sourced from handbooks and rules-of-thumb enabling engineers to check different attributes of the design within the modeling environment.

Most of these commercial DFM systems have a fixed rules database, to which any addition is typically made by the software vendor. Tools such as NX's Checkmate permit adding custom rules but the interface is not intuitive

[1] http://www.cs.ox.ac.uk/activities/machlearn/Aleph/
[2] http://www.doc.ic.ac.uk/~shm/progol.html
[3] http://www.ahlgren.info/research/atom/
[4] http://www.swi-prolog.org/
[5] http://dfmpro.geometricglobal.com/
[6] http://www.solidworks.com/
[7] http://www.plm.automation.siemens.com/en_us/products/nx/
[8] http://www.dfma.com/
[9] http://www.apriori.com/

and the programming effort required to carry out additions is tedious.

Integrated DFM Learning Platform

We have developed a DFM platform (Rangarajan et al. 2013) that provides manufacturability analysis and design feedback based on semantic technologies. We have extended this semantic DFM platform so that we can automatically generate feature recognition rules in the manufacturability domain. In our target domain, it is critical that our platform be integrated with the standard domain specific CAD tool. Consequently, in our integrated platform, the semantic model is integrated with NX environment. This integration is provided by Java code that mediates between Jena objects and NX objects. The three components in our platform architecture are therefore NX CAD tool, Java processing and a semantic model, and the control and data flow is as shown in Fig. 1. The NX CAD tool provides geometry and topological information, and it also provides visualization of the part design. The semantic model is an OWL model and it provides a graph representation of the geometry and the ability to encode feature recognition and manufacturability rules. Java controls logic of events and it is also the bridge between the CAD and the OWL. Since OWL and CAD APIs are not (currently) compatible, Java is necessary to realize the communication.

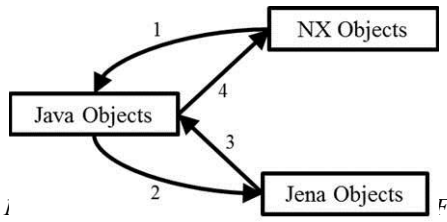

Figure 1. Flow.

NX

One of our goals is to make our platform as CAD tool agnostic as possible. For this reason we work with the B-rep[10] representation of the part. B-rep is a popular representation that uses faces, edges, vertices and their properties to describe a part. The Java processing uses NX APIs to access the B-rep representation of the part and make them available to the semantic model.

Semantic Model

We represent the semantic model using the Semantic Application Design Language (SADL)[11] (Crapo and Moitra 2013) which is an English-like language for representing ontologies. The SADL tool, which is available as a plugin to Eclipse, automatically translates statements in SADL to OWL. SADL uses Apache Jena[12] as the default inference engine. SADL also enables us to represent rules which are then translated to Jena rules. We translate the rules learnt using Aleph into SADL in order to enable querying over the OWL + rules integrated framework using SPARQL[13]. A portion of the semantic model in SADL is shown in Fig. 2.

```
AdjacencyType is a class,
  must be one of {TANGENT, CONVEX,
                  CONCAVE, UNKNOWN}.
Vertex is a type of AbstractSADLnx,
   described by connectedEdges with values of type AbstractEdge.

AbstractEdge is a type of AbstractSADLnx,
   described by endpoint with values of type Vertex,
   described by connectedTo with values of type AbstractEdge,
   described by edgeAdjacencyType
                with a single value of type AdjacencyType,
   described by connectedFaces with values of type AbstractFace,
   described by edgeAdjacencyAngle
                with a single value of type double.

connectedFaces of AbstractEdge has exactly 2 values.

{Circular, Elliptical, Intersection, Linear, Spline, SP_Curve}
are types of AbstractEdge.

AbstractFace is a type of AbstractSADLnx,
   described by edge with values of type AbstractEdge,
   described by adjacentFace with values of type AbstractFace,
   described by faceAdjacencyType
                with a single value of type AdjacencyType.
faceAdjacencyType of AbstractFace has default UNKNOWN.

{Blending, Conical, Cylindrical, Parametric, Planar, Spherical,
 Surface_Of_Revolution}
are types of AbstractFace.
```

ILP Set Up

For ILP we need to provide background knowledge; and in our platform we base it directly on the semantic model. So, the class *AbstractFace* in the semantic model is represented using the predicate *face*; and the property *adjacentFace* that has *AbstractFace* as both domain and range in the semantic model is represented as *adjacentface(+face,-face)*. The background knowledge also allows us to select what predicates can be used in the construction of generated rules, thereby providing flexibility and directing rule generation process. The background knowledge is shown in Fig 3.

[10] https://en.wikipedia.org/wiki/Boundary_representation
[11] http://sadl.sourceforge.net/
[12] https://jena.apache.org/
[13] http://www.w3.org/TR/rdf-sparql-query/

```
% Hypothesis declaration; feature to be learned
:- modeh(1, new_feature(+face)).

% Background knowledge declaration
:- modeb(4,type(+face,#fetype)).
:- modeb(4,adjacentface(+face,-face)).
:- modeb(4,type(+edge,#fetype)).
:- modeb(4,edge(+face,-edge)).
:- modeb(*,connectedto(+edge,-edge)).
:- modeb(*,connectedfaces(+edge,-face)).
:- modeb(4,edgeadjacencytype(+edge,#edgeadjacencytype)).
:- modeb(4,faceadjacencytype(+face,#faceadjacencytype)).
:- modeb(4,closed(+face)).
:- modeb(4,concave(+face)).
:- modeb(4,notconcave(+face)).

% what can be used in generated rule
:- determination(new_feature/1,type/2).
:- determination(new_feature/1,adjacentface/2).
:- determination(new_feature/1,edge/2).
:- determination(new_feature/1,connectedto/2).
:- determination(new_feature/1,connectedfaces/2).
:- determination(new_feature/1,edgeadjacencytype/2).
:- determination(new_feature/1,faceadjacencytype/2).
:- determination(new_feature/1,concave/1).
:- determination(new_feature/1,notconcave/1).

% type definitions, fe == face or edge
fetype(blending). fetype(conical). fetype(cylindrical).
fetype(parametric). fetype(planar).
fetype(spherical). fetype(surface_of_revolution).

fetype(circular). fetype(elliptical). fetype(intersection).
fetype(linear). fetype(spline). fetype(sp_curve).

% provide local instance data for
% positive and negative examples from the part
```

Figure 3. Fragment of ILP Background Knowledge.

The background knowledge also includes instance data which is generated as follows: the platform writes out the instance data for the part in N-triples[14] format and then uses SWI-Prolog to ingest and translate it into literals based on the predicates declared in Fig 3. For running ILP, we retain just local information for the positive and negative examples so that the rules learned are general enough to be applicable across different parts. For example, if a (positive or negative) example identified was face *f0*, then the instance data supplied to ILP consists of *f0* and all its properties including edges of *f0* and adjacent faces of *f0*. It also includes properties of the edges of *f0* and the properties of adjacent faces of *f0*. If *f0* has an adjacent face *f1*, and *f1* has an adjacent face *f2* (and *f2* is not an adjacent face of *f0*) then *f2* is not included in the instance data related to *f0*. The generation of this local information is also done as part of the processing in SWI-Prolog.

[14] http://www.w3.org/TR/n-triples/

If f0 is a positive or negative example, it is represented as *new_feature(f0)*. By convention, all positive examples are placed in a file with extension ".f" (f for fact) and all negative examples are placed in a file with extension ".n" (n for negative).

The results obtained by running the generated rules are integrated into the platform so that they can be visualized using NX and the Integrated DFM Learning Platform as shown in Fig. 4. We can iterate over different positive and negative examples to generate learned rules. Once we are satisfied with the rule that is learned for a concept then we encode that rule in the semantic model. It is then available as a domain concept for any subsequent iterations of ILP to learn additional concepts.

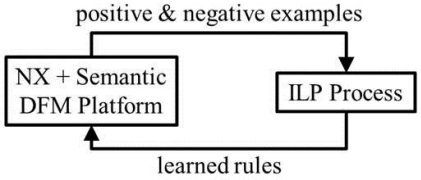

Figure 4. Integrated DFM Learning Platform.

Semantic DFM Platform

The learned feature recognition rules are folded back into the semantic DFM platform where manufacturability rules are defined for the various features. The learned rules are persisted and maintained as part of the DFM platform.

Illustrative Examples

We have used the Integrated DFM Learning Platform to work with large industrial designs like compressor casings and have generated feature recognition rules for pad fillet, tapers, spot faces, flange faces etc. Here we will illustrate this platform via 2 examples.

Example 1

In this example we will illustrate the working of the developed Integrated DFM Learning Platform on a very simple part as shown in Fig. 5. As can be seen in Fig. 5, 4 holes are visible, one is a through hole, another is a blind hole with a flat bottom and remaining 2 holes are blind holes each with a conical bottom. The goal is to generate a rule that identifies blind holes with a conical bottom. The user selects 2 positive instances of the feature to be recognized, which are highlighted as shown in Fig. 5a. In general, it is not necessary to select all positive instances present and that will be illustrated in an example later on.

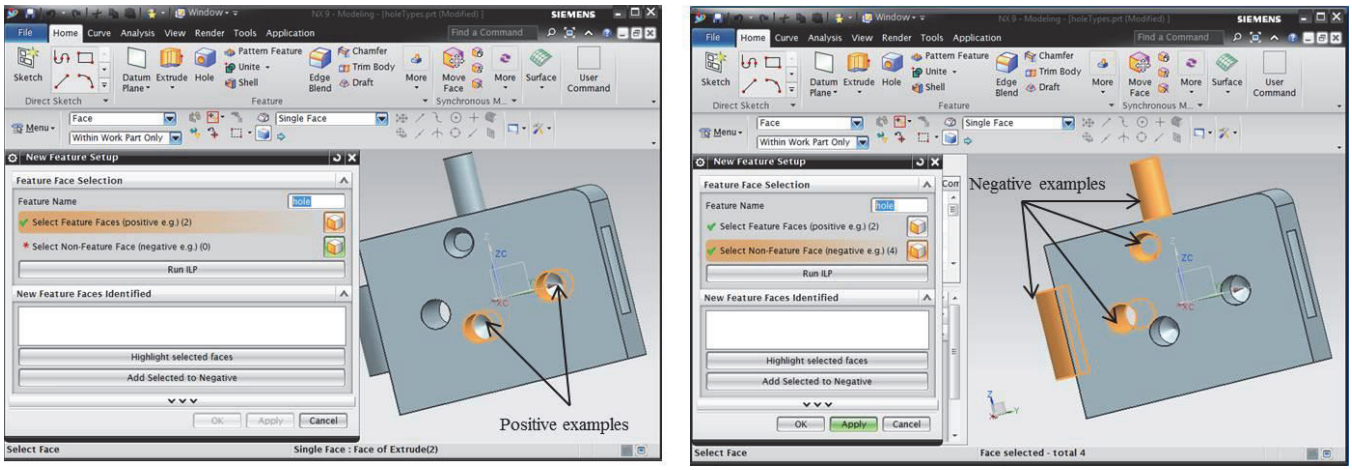

Figure 5. Selecting Examples for Blind Holes with Conical Bottom - Selections Displayed in Orange and with Arrows. (a) Positive Examples. (b) Negative Examples.

The user then selects 4 negative examples, which are highlighted as shown in Fig. 5b. The Integrated DFM Learning Platform then generates appropriate instance data as explained in previous section and calls Aleph. In this example, Aleph learns one rule as follows:

new_feature(A) :- adjacentface(A,B), type(B,conical).

The Integrated DFM Learning Platform shows the result of applying this generated rule on the entire part in Fig. 6. In this case, the generated rule identifies 4 faces that satisfy the rule and 2 of these are the positive examples that were supplied and the other 2 are Planar faces. These 2 Planar faces are listed in Fig. 6 and the user can select any subset of them and visualize them on the part. Fig. 6 shows these 2 Planar faces highlighted.

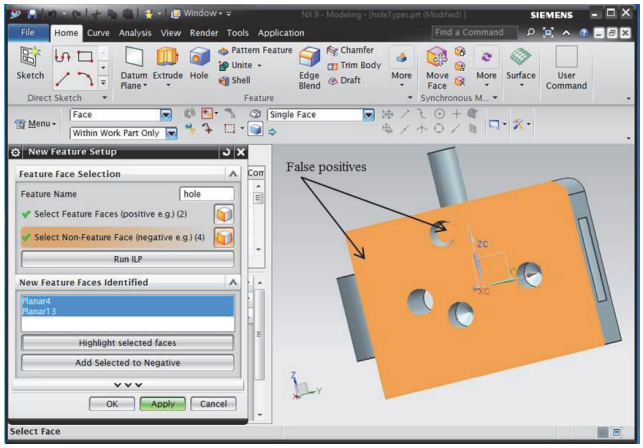

Figure 6. Genarated Rule Evaluation – False Positives Displayed in Orange and with Arrows.

Since these Planar faces shown in Fig. 6 are not holes, the user can add both of these as additional negative instances and re-run the ILP rule generation process. With the new set of examples (2 positive instances and 6 negative instances), Aleph learns a new rule as follows:

new_feature(A) :- type(A,closedcylindrical),
adjacentface(A,B), type(B,conical).

This new rule identifies only the original positive examples as holes. We should point out that for this illustrative example we used a very simple part, if we use a more complicated part, the generated rule may change. The rule generated here simply separates the positive examples from negative examples; and in fact this rule will match blind holes with conical bottom as well as some types of protrusions. If we want to generate a rule that only identifies blind holes, then we would need to use an appropriate part and select appropriate positive and negative examples.

Another feature of this approach is that we can select what properties can be used in the generated rule. So if we drop the property “closedcylindrical” from being available for rule generation, then the rule generated is as follows:

new_feature(A) :- adjacentface(A,B),
type(B,conical), closed(A).

Example 2

In this example, we consider an actual part that is both large and complicated. The part is the top-half casing of an aircraft engine and the part model consists of 4743 faces and 10,126 edges. We had previously manually authored rules to detect pad fillets and had 4 rules to handle various sub-cases. Here we will focus on generating just one of these 4 rules, which is a rule to recognize if a Surface_Of_Revolution face is part of a pad fillet or not. Also, since the top-half casing part has a huge number of pad fillets, we decided that we could work with just a slice of the entire part for this rule generation exercise. This slice,

called target part here, is still large and has 823 faces and 2067 edges. The rule that we had previously manually authored is shown in Fig. 7. By running this rule on the target part we get 41 pad fillets which will serve as our ground truth.

```
Rule FindPadFillet3
if
  f0 is a Surface_Of_Revolution
  f0 has edge e1
  e1 has edgeAdjacencyType TANGENT
  e1 is a Circular
  f0 has edge e2
  e1 != e2
  e2 has edgeAdjacencyType TANGENT
  e2 is a Circular
  e2 has connectedFaces f2
  f2 is a Cylindrical
  concave of f2 is false
  e1 has connectedFaces f1
  f0 !=f1
  f1 != f2
  // f1 and f2 do not share any vertices
  lv1 is list(f1,edge,xe1, xe1,endpoint,v,
          f2,edge,xe2, xe2,endpoint,v)
  listLength(lv1) = 0
  fillet1=getInstance(PadFillet, featureFace,f0,
                      bottomFace,f2, bottomEdge,e2)
then
  otherFace of fillet1 is f1
  calculateAngle of fillet1 is false
  featureName of fillet1 is "Pad Fillet".
```

Figure 7. Manually Authored Rule for Pad Fillet.

For the first iteration of rule generation in the Integrated DFM Learning Platform, we selected 12 positive and 10 negative examples – note that we did not select all positive instances. The rule that was generated is as follows:

```
new_feature(A) :- facetype(A,surface_of_revolution),
                  adjacentface(A,B),
                  notconcave(B).
```

When this rule is evaluated on the target part, it finds 48 instances of pad fillets. Of these 48 instances, 41 are same as our ground truth and the remaining 7 are not pad fillets so we need to iterate in order to refine the rule.

In the second iteration, we added the 7 incorrect instances as additional negative instances; so this run used 12 positive and 17 negative instances. The new rule that was generated is as follows:

```
new_feature(A) :- facetype(A,surface_of_revolution),
                  adjacentface(A,B),
                  notconcave(B),
                  adjacentface(A,C),
                  facetype(C,planar).
```

The evaluation of this rule matched exactly the ground truth and so we can stop iterating the rule generation process. Note that this automatically generated rule is much simpler than the one that was manually authored.

In general, if ground truth is not available then the rule can be evaluated on additional parts to see what feature instances are identified or a SME can evaluate the rule.

Results

The Integrated DFM Learning Platform described in this paper has been implemented and has been used in automatically generating feature recognition rules. We have been successful in generating rules for simple features like depression and protrusion (Brousseau, Dimov, and Setchi 2008), we have also shown that we can generate rules for recognizing more complex features that may include numerical attributes (e.g. spot face rule). Some of the features for which we have successfully generated rules are as follows

- taper
- pad fillet
- spot face
- flange face
- candidate angled face (face that should be converted to angled faces so as to reduce manufacturability cost)

For these features, the rules generated had accuracy 1. Further for taper and spot face features it took a single iteration to generate the final rule. For pad fillet and flange face it took 2 iterations to achieve the final rule. For the candidate angled faces we had 3 small sample parts and since we process one part at a time, it took 3 iterations. Note also that the number of iterations needed is dependent on how many and which positive and negative examples are selected.

Related Work and Conclusions

Since a vast amount of domain knowledge has already been captured in text, considerable effort has been made in extracting this written knowledge into formal models, see (Wong, Liu, and Bennamoun 2012) for a survey of various approaches. Most of this effort has been in extracting concepts and relationships between the concepts and representing it in a semantic model. There has also been work in extracting rules from manufacturing handbooks (Kang et al. 2015).

In this paper we have considered how we can automate the capture of domain knowledge by applying Inductive Logic Programming to positive and negative instance data. We have shown this by developing an Integrated DFM Learning Platform for generating feature recognition rules from complex parts. This platform is currently in use for

providing early manufacturability feedback in an industrial setting.

Acknowledgments

The authors would like to thank Christine Furstoss, Global Technology Director for Materials and Manufacturing & Materials Technologies at GE Global Research, for supporting this project. The authors also thank Terri Jia for helping to develop the code base and rules used for testing ILP techniques; and to Mike Graham and Dean Robinson for their comments and feedback.

References

Babic, B.; Nesic, N.; and Miljkovic, Z. 2008. A Review of Automated Feature Recognition with Rule-Based Pattern Recognition. *Computers in Industry* 59(4): 321-337.

Bratko, I.; and Muggleton, S. 1995. Applications of Inductive Logic Programming. *Communications of the ACM* 38(11): 65-70.

Brousseau, E.; Dimov, S.; and Setchi, R. 2008. Knowledge Acquisition Techniques for Feature Recognition in CAD Models. *Journal of Intelligent Manufacturing* 19(1): 21-32.

Chen, D. L.; and Mooney, R.J. 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 859-86.

Crapo, A.; and Moitra, A. 2013. Toward a Unified English-like Representation of Semantic Models, Data and Graph Patterns for Subject Matter Experts. *International Journal of Semantic Computing* 7(3): 215-236.

Faruquie, T. A.; Srinivasan, A.; and King, R.D. 2013. Topic Models with Relational Features for Drug Design. *Inductive Logic Programming*, Springer Berlin Heidelberg, 45-57.

Kang, S.; Patil, L.; Rangarajan, A.; Moitra, A.; Jia, T.; Robinson, D.; and Dutta, D. 2015. Extraction of Manufacturing Rules from Unstructured Text Using a Semantic Framework. *ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing* 8(4): 295–318.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Computing Journal* 13(3-4): 245-286.

Phelan, K.; Wilson, C.; and Summers, J.D. 2014. Development of a Design for Manufacturing Rules Database for Use in Instruction of DFM Practices. *ASME 2014 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers.

Radhakrishnan, R.; Amsalu, A.; Kamran, M.; and Nnaji, B. O. 1996. Design Rule Checker for Sheet Metal Components using Medial Axis Transformation and Geometric Reasoning. *Journal of Manufacturing Systems* 15(3): 179-189.

Rangarajan, A.; Radhakrishnan, P.; Moitra, A.; Crapo, A.; and Robinson, D. 2013. Manufacturability Analysis and Design Feedback System Developed using Semantic Framework. In *Proceedings of the ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, American Society of Mechanical Engineers.

Wang, Q. 2012. Developing a Computational Framework for Explanation Generation in Knowledge-Based Systems and its Application in Automated Feature Recognition. Ph.D. Diss. RMIT University.

Wong, W.; Liu, W.; and Bennamoun, M. 2012. Ontology Learning from Text: A Look Back and into the Future. *ACM Computing Surveys* (CSUR), 44(4), 20.

Wielemaker, J.; Schrijvers, T.; Triska, M.; and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12(1-2): 67-96.