

A NEW ILP SYSTEM FOR MODEL TRANSFORMATION

BY EXAMPLES

BY

HAMDI ALI AHMED AL-JAMIMI

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

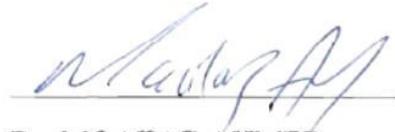
MAY 2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

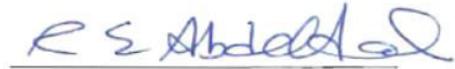
This thesis, written by **HAMDI ALI AHMED AL-JAMIMI** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**



Dr. MOATAZ AHMED
(Advisor)



Dr. ADEL AHMED
Department Chairman



Dr. RADWAN ABDEL-AAL
(Member)



Dr. Salam A. Zummo
Dean of Graduate Studies



Dr. SHOKRI SELIM
(Member)

6/7/15

Date



Dr. MOHAMMAD ALSHAYEB
(Member)



Dr. MAHMOUD ELISH
(Member)

© Hamdi Ali Al-Jamimi

2015

DEDICATION

To my parents for their love, support and prayers

To my wife for being my inspiration

ACKNOWLEDGMENTS

First and foremost, all praise and thanks are due to Almighty ALLAH for giving me the chance to study as far as this and for the strength not to give up.

I acknowledge all the support given by King Fahd University of Petroleum and Minerals during my graduate studies and during this research.

I would like to express my deepest gratitude to my advisor Dr. Moataz Ahmed for all his valuable guidance, unlimited help and continuous support not only during this work but through- out the entire degree. My gratitude is also due to the members of the doctoral committee for their attentions and enlightening comments.

I would like to express my gratefulness to my parents. Even though they have not directly influenced the work itself, they are the secret of my success and happiness. I would not be where I am today without their prayers, love and encouragement. I also want to thank to my dear wife for all her patience, friendship and tolerance.

Last, but not least, I am grateful to all my friends and family members who supported me in many ways during this study. |

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES	XII
LIST OF FIGURES	XIII
LIST OF ABBREVIATIONS	XV
ABSTRACT	XVII
ملخص الرسالة	XIX
CHAPTER 1 INTRODUCTION	1
1.1 Research Context	1
1.1.1 Model Driven Engineering	1
1.1.2 Software Reuse	3
1.1.3 Knowledge Reuse	4
1.2 Motivation and Objectives	5
1.3 Dissertation Contributions	8
1.4 Dissertation Organization	9
CHAPTER 2 BACKGROUND	12
2.1 Model Transformations	12
2.1.1 Modeling Tools and Transformation Languages	15
2.1.2 Transformations by Examples	16
2.1.3 Transformations by Demonstration	17
2.2 Learning Theory	18

2.2.1	Learning: A Continuous Process.....	19
2.2.2	Inductive Learning	20
2.3	Inductive Logic Programming	21
2.3.1	Basic ILP Concepts	22
2.3.1.1	Illustrative Example	23
2.3.1.2	Mode Declarations.....	26
2.3.1.3	Bias	27
2.3.1.4	Structuring the Hypothesis Space	28
2.3.2	ILP Dimensions	30
2.3.2.1	Batch Learning and Incremental Learning	30
2.3.2.2	Interactive Learning and Non- Interactive Learning	31
2.3.2.3	Single Predicate Learning and Multiple Predicates Learning	32
2.3.3	ILP Systems.....	33
2.3.3.1	Top-Down ILP Systems.....	33
2.3.3.2	Bottom-Up ILP Systems.....	36
2.3.4	Potential Advantages of ILP.....	39
2.4	Rule-Based Expert Systems	41
2.4.1	An Overview	41
2.4.2	Java Expert System Shell	42
CHAPTER 3 LITERATURE SURVEY.....		44
3.1	Transformation from Analysis to Design	44
3.2	Model Driven Development	46
3.2.1	CIM-to-PIM.....	47
3.2.2	PIM-to-PSM	50
3.3	Towards Model Transformation Generation	52

3.3.1	Model Transformation by Example Approaches	53
3.3.2	Model Transformation by Demonstration	57
3.3.3	Analogy-based MTBE.....	58
3.4	Using Prolog Rules to Detect Software Design Patterns	62
3.5	Existing ILP Applications.....	64
CHAPTER 4 KNOWLEDGE REPRESENTATION		68
4.1	Knowledge Representation: An overview	68
4.2	Software Artifacts: Formal Representation	69
4.2.1	Models Constructs Predicates	70
4.2.2	Relationship Predicate.....	71
4.3	Assessing Prolog: Strengths and Weaknesses.....	73
4.3.1	Strengths.....	73
4.3.2	Weaknesses.....	74
CHAPTER 5 THE RESEARCH PROBLEM AND PROPOSED SOLUTION		77
5.1	Research Questions	77
5.2	Solution Approach	79
5.3	Preprocessing.....	81
5.3.1	Background Knowledge and Examples Files Creation	81
5.3.2	Grouping of the Positive Examples	83
5.4	Rules Induction and Generalization	85
5.5	Rules Application and Evaluation.....	87
5.5.1	JESS Rules Translation	87
5.5.2	Measuring Performance of the Induced Rules	88
5.5.3	Application and Evaluation using a GA-based procedure.....	90

5.5.3.1	Procedure 1: Individual Rule Application on All Source Models:	91
5.5.3.2	Procedure 2: GA-based Rules Application on an Individual Learning System	91
5.6	Rules Validation and Refinement	93
5.7	Post-processing	95
CHAPTER 6 INDUCTION USING ALEPH SYSTEM		96
6.1	ALEPH Overview	96
6.1.1	ALEPH Input Requirements.....	97
6.1.1.1	Background Knowledge	98
6.1.1.2	Positive and Negative Examples.....	100
6.2	Representation of Model Transformation Problem.....	101
6.2.1	Sample Transformation Tasks.....	101
6.2.1.1	Packaging Class Diagram	101
6.2.1.2	Introducing Façades.....	103
6.2.2	Problem and Solution Representations	104
6.3	Experiments Setup	109
6.3.1	Datasets	109
6.3.2	ALEPH Settings	111
6.4	Experimental Results / Quantitative Validation	111
6.4.1	Performance of the Packaging Rules	112
6.4.2	Validation of Induced Rules	115
6.4.3	Accuracy of Façades Rules	116
6.5	Discussion	118
6.5.1	Open Issues	119
6.5.2	The Current ILP Systems in MDD Context	121

CHAPTER 7 THE PROPOSED ILP SYSTEM.....	125
7.1 An overview	125
7.2 MTILP Generic Algorithm	127
7.2.1 Incremental Learning.....	129
7.3 Hypothesis Induction Process	130
7.3.1 Hypothesis Space Construction	130
7.3.2 Bottom Clauses Construction	132
7.3.3 Bottom Clauses Substitution	135
7.4 Search for the Hypothesis	138
7.5 Learning from Negative Examples	143
7.6 Comparison with Prominent ILP Systems	145
CHAPTER 8 EMPIRICAL EVALUATION.....	149
8.1 Experiments and Results	150
8.1.1 Packaging Transformation Rules.....	150
8.1.1.1 Rules Induction Using Learning Examples	151
8.1.1.2 Evaluation of the Induced Rules.....	153
8.1.1.3 Rules Validation	155
8.1.2 Introducing Façade Rules.....	157
8.1.2.1 Rules Induction: Incremental Learning.....	159
8.1.2.2 Rules Evaluation	161
8.1.2.3 Rules Validation	163
8.2 More Model Transformation Tasks	164
8.2.1 The Blob Anti-pattern	164
8.2.2 Splitting a Class into Partial Classes	168
8.3 Application of MTILP in Other Contexts	170

8.3.1	Learning Transitive Rules.....	170
8.3.2	Patrick Winston's arch problem.....	174
8.4	MTILP vs. ALEPH.....	175
CHAPTER 9 CONCLUSIONS AND FUTURE WORK		180
9.1	Major Contributions.....	180
9.2	Limitations and Threats to Validity	181
9.3	Future Work.....	182
REFERENCES.....		184
VITAE.....		197

LIST OF TABLES

Table 1: Generalization and Specification	30
Table 2: Characteristics of various ILP systems.....	38
Table 3: Summary of CIM to PIM transformation approaches	49
Table 4: Generating Model Transformation Rules from Examples.....	61
Table 5: Summary of logic-based approaches for design patterns detection.....	64
Table 6: Predicate to define the artifacts types	70
Table 7: Predicate to describe artifacts that have other artifacts	71
Table 8: Predicate to describe relations between artifacts	72
Table 9: Performance Measurements.....	89
Table 10: Packaging problem representation written in ALEPH (i.e., Prolog Syntax)	105
Table 11: Introducing Facade - Problem representation written in ALEPH (i.e., Prolog Syntax).....	106
Table 12: The Solution Representation.....	107
Table 13: The datasets Statistics	109
Table 14: Description of the used datasets	110
Table 15: Samples of the induced transformation rules using ALEPH- Packaging Problem	112
Table 16: Samples of the induced transformation rules using ALEPH- Introducing Facade	112
Table 17: A comparisons of ILP systems	147
Table 18: Samples of the induced transformation rules - Packaging transformation task	151
Table 19: Samples of the induced transformation rules - Introducing Façade	160
Table 20: Rules induced using ALEPH and MTILP systems.....	167
Table 21: The representation of Michalski's train problem	172
Table 22: Induced rules - Arch Problem.....	175

LIST OF FIGURES

Figure 1: High level view of proposed process	4
Figure 2: Model transformation process	13
Figure 3: Types of MDE transformations	14
Figure 4: A form of the transformation example	17
Figure 5: The application of inductive learning.....	21
Figure 6: Example of a family tree	23
Figure 7: Representation of the background knowledge	24
Figure 8: Representation of the positive and negative examples.....	25
Figure 9: Snapshot of modes declarations using ALEPH.....	27
Figure 10: Basic structure of an Expert System.....	42
Figure 11: A sample of JESS rule declaration	43
Figure 12: Architecture of the proposed transformation system	80
Figure 13: Life cycle of the input files.....	82
Figure 14: Example of a chromosome representation	92
Figure 15: Examples of rules to be subsumed	95
Figure 16: The UML class diagram for analysis-design pair (Source model).....	102
Figure 17: The UML class diagram for analysis-design pair (Target model)	102
Figure 18: Example of Introducing Facade (Source Model)	103
Figure 19: Example of Introducing Facade (Target Model).....	104
Figure 20: Sample rule implemented by JESS rule engine.....	108
Figure 21: Performance Measures - Rules application on batched learning systems...	113
Figure 22: Rules Performance - Learning systems one by one	114
Figure 23: Scores assigned to each rule based on their performance.	114
Figure 24: Average of accuracy measures (validation systems) with different number of rules.	115
Figure 25: Average of accuracy measures (validation systems) with random selected rules.	116
Figure 26: Accuracy measures - Learning Systems.....	117
Figure 27: Accuracy Measures - Validation Results	117
Figure 28: Example to show the drawback of the derived rule	121
Figure 29: Flowchart of MTILP Algorithm.....	128
Figure 30: The Induction Process Flowchart.....	131
Figure 31: Ground most-specific clause for example daughter /2 when L =1.....	134
Figure 32: Ground most-specific clause for example daughter/2 when L =3.....	135
Figure 33: A substituted most-specific clause	136
Figure 34: The construction of the search space when inducing daughter relation.....	141
Figure 35: Snapshot of the search tree when inducing daughter relation	142
Figure 36: Snapshot of the final hypothesis using MTILP system.....	143

Figure 37: Learning from positive and negative data	144
Figure 38: Accuracy measures of the induced rules (all the learning systems).....	152
Figure 39: Ranking induced rules based on their Accuracy measures	153
Figure 40: Evaluating the Induced Rules Using Learning Systems	154
Figure 41: Percentage of the application frequency that rules selected to give best result.....	154
Figure 42: The set of 24 rules ranked based on their application frequency	155
Figure 43: Selection of the rules for validation	156
Figure 44: Selection of the rules randomly for validation	157
Figure 45: New artifacts when introducing façade interface	158
Figure 46: The positive examples related to introducing a façade	159
Figure 47: Accuracy measures for the induced rules.....	161
Figure 48: Frequency of the rules application – Façade Transformation	162
Figure 49: Introducing Façade rules ranked based on their application times	162
Figure 50: Validation results – various combinations	163
Figure 51: The Blob anti-pattern (Source model).....	164
Figure 52: The Blob anti-pattern (Target model)	165
Figure 53: Snapshot of the class and sequence diagram used as learning example	166
Figure 54: The Blob anti-pattern - Problem Representation.....	167
Figure 55: A Example of splitting class– source and target models.....	168
Figure 56: A Snapshot of the background knowledge - Splitting class example	169
Figure 57: A snapshot of the presented positive examples.....	169
Figure 58: Sample of the induced rule -Splitting Class	170
Figure 59: Michalski s original ten trains	171
Figure 60: Example of Patrick Winston's arch problem.....	174
Figure 61: Introducing packages to the class diagram.....	177
Figure 62: Introducing Façade design pattern	178
Figure 63: MTILP versus ALEPH summary	179

LIST OF ABBREVIATIONS

ALEPH	:	A Learning Engine for Proposing Hypotheses
ATL	:	ATLAS transformation language
CIM	:	Computation Independent Model
CSP	:	Constraint Satisfaction Problem
CWA	:	Closed World Assumption
ILP	:	Inductive Logic Programming
MDA	:	Model Driven Architecture
MDD	:	Model Driven Development
MDE	:	Model Driven Engineering
ML	:	Machine Learning
MPL	:	Multiple Predicates Learning
MT	:	Model Transformation
MTBD	:	Model Transformation By Demonstration
MTBE	:	Model Transformation By Example
MTILP	:	Model Transformation by Inductive Logic Programming
PIM	:	Platform Independent Model

PSM	:	Platform Specific Model
OMG	:	Object Management Group
RLGG	:	Relative Least General Generalization
SPL	:	Single Predicate Learning

ABSTRACT

Full Name : [Hamdi Ali Ahmed Al-Jamimi]
Thesis Title : [A New ILP System for Model Transformation By Examples]
Major Field : [Computer Science and Engineering]
Date of Degree : [May 2015]

[The emergence of Model-Driven Engineering (MDE) has changed the focus of software development from code to models, and raised the need for model transformations. Model Transformation requires transformation rules provided by the experts in the underlying domain. However, it would be much easier for human experts to provide examples of transformations rather than writing a set of transformation rules. Model Transformation By Example (MTBE) is a novel approach in MDE. The rationale behind this approach is to utilize existing knowledge represented by source and target models of previously developed systems. Such knowledge can be utilized to derive transformation rules to be applied in future system developments. Discovering and formalizing the desired transformation rules can be automated by employing machine learning (ML) algorithms. Inductive Logic Programming (ILP) represents a highly applicable ML technique in this context. It utilizes the power of ML and the capability of logic programming to induce valid hypotheses from given examples.

In this dissertation, we developed an ILP-based transformation system that can acquire and reuse the existing knowledge (pairs of source-target models) to generate transformation rules, and then utilize them in the future developments. In particular, we address the problem of transforming analysis models into design models. Our approach

can be classified as a MTBE approach; it utilizes the available knowledge manifested in the form of examples to build the transformation model.

Moreover, this work offers two contributions. As a first contribution, we introduce a detailed architecture of ILP-based transformation system. The proposed system consists of three components employed to induce, evaluate, validate and refine the transformation rules. The engine used to induce the rules is an ILP system. We employ an existing ILP system, called ALEPH system, to drive the rules and we use the JESS language and engine to make the rules executable. We also study the limitations of the ALEPH, and how we better overcome the discovered limitations.

The second contribution is in the form of a new ILP system, MTILP (Model Transformation using Inductive Logic Programming), to induce MDE transformation rules. Experiments show the ability of MTILP to overcome the limitations of ALEPH within the context of MDE transformation. |

ملخص الرسالة

الاسم الكامل: حمدي علي أحمد الجميمي

عنوان الرسالة: نظام برمجة المنطق الاستقرائي لتحويل النماذج باستخدام الأمثلة

التخصص: علوم وهندسة الحاسب الآلي

تاريخ الدرجة العلمية: مايو 2015

بتطوير هندسة البرمجيات ظهرت هندسة النماذج (MDE) واصبح الاهتمام بالنموذج اكثر من الرمز، وبالتالي زادت الحاجة لتحويل النماذج من صيغة إلى أخرى. ويتطلب تنفيذ تحويل النموذج قوانين يمكن أن يقدمها خبراء في ذات المجال. ولكن من الأسهل للخبراء تقديم أمثلة من التحويلات بدلا من كتابة مجموعة من القوانين. مؤخرا ظهر اسلوب جديد لتحويل النماذج يسمى نموذج التحويل حسب المثال (MTBE) ويعتمد على استنباط المعرفة المضمنة في أمثلة تتكون من نماذج المصدر والهدف. هذه المعرفة يمكن استخدامها لاستخلاص قوانين التحول ليتم تطبيقها في تطوير النظام في المستقبل. اكتشاف وإضفاء الطابع الرسمي على القوانين المطلوبة يمكن أن يكون آليا عن طريق استخدام خوارزميات آلة التعلم (ML). برمجة المنطق الاستقرائي (ILP) تمثل تقنية ML تنطبق إلى حد كبير في هذا السياق، حيث انه يوظف قوة ML وايضا القدرة على البرمجة المنطقية لاستنباط فرضيات من الأمثلة المعطاة.

في هذه الأطروحة، قمنا بتطوير نظام تحويل يعتمد على ILP والذي يستخدم المعارف المتاحة في شكل أمثلة لبناء نموذج التحويل. يقوم النظام باستنباط قوانين التحويل من الامثلة المتوفرة وإعادة استخدامها في التطبيقات المستقبلية. وعلاوة على ذلك، فإن هذا العمل يقدم مساهمتين. المساهمة الاولى تركزت على تقديم بنية تفصيلية لنظام التحويل المعتمد على ILP والمكون من ثلاثة مكونات رئيسية لاستنباط قوانين التحويل وتقييمها والتحقق من صحتها. بواسطة احد الانظمة المتاحة قمنا بدراسة أوجه القصور في أنظمة ILP الحالية في سياقنا، وكيف نتغلب على مشكلاتها.

وتعتبر المساهمة الثانية هي تقديم نظام ILP جديد بإسم نموذج التحويل باستخدام برمجة المنطق الاستقرائي (MTILP). وقد تبين من خلال التجارب قدرة MTILP للتغلب على اوجه القصور في أنظمة ILP الحالية في سياق

تحويلات الـ MDE.

CHAPTER 1

INTRODUCTION

Model Driven Engineering (MDE) considers the models as the primary artifacts in the software development process. It provides support for the creation and the transformation of the various types of models.

1.1 Research Context

In this section we discuss three dimensions of the problem context.

1.1.1 Model Driven Engineering

Model Driven Engineering is a methodology of software development that provides supports for creating, editing models, and transforming models to other models or programs [1]. The usage of models helps improving the productivity in two ways: i) first, model reuse through the maximization of compatibility between systems; ii) second, standardization through the simplification of the process design. Hence, MDE gains growing interest from the industry, which considers it a possible solution for the ever growing quality factors, performances, and maintainability. MDE involves various principles including those based on the separation of business logic from platforms technology [2, 3].

Three types of models can be distinguished in the model-driven paradigm: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). CIM focuses on the domain a higher level of abstraction instead of showing the details of the system structure. CIM is then transformed to the subsequent model, PIM. PIM is offer design decisions without considering the underlying platform or any other technical considerations. PIM is further transformed into .PSM. PSM includes the technical considerations and the underlying platform [4].

Moreover, in MDE, several kinds of models can be handled and the models may be derived from each other, in an automated way. That is, the artifacts in a particular model can be utilized to produce new model artifacts for the same system, which is known as the *Model Transformation* concept [5].

A model transformation is defined as a methodology that receives as input a model in a source language and, in turn, it produces another model in a target language as output. The transformation is formed around a set of relationships that can be formalized between source model artifact(s) and target artifact(s). Implementing a model transformation requires a strong knowledge about Model-Driven Development (MDD) paradigm including the meta-models and the environment of the model transformation. The meta-model defines the structure, i.e., concepts and their relationships that can be used to compose a model.

Model Transformation requires identifying the needed transformation rules by the domain experts. The transformation rule can be defined as a description of how one or more artifacts presented in the source language can be transformed into one or more

artifacts in the target language. However, it is not a trivial task to author and define model transformation rules. Thus, it is much easier for the expert to present transformation examples that consist of source-target pairs instead of providing a complete set of transformation rules.

Model Transformation By Example (MTBE) is an approach that aims at learning a model transformation from the provided examples, where each example represents a pair of the transformation source and target models. In practical, a learning approach can be used to deduce the transformation rules from the available set of examples.

Based on that, there is a need for intelligent tools that utilize the available pairs of source/target models to learn the relations linking the source model artifacts to their corresponding artifacts in the target model. The purpose of these tools is not only to implement the intended transformation model, but also to provide support for more complex reasoning abilities and exploration of traces between the source and target models spaces.

1.1.2 Software Reuse

Software reuse is defined as the process of building new software systems by the use of artifacts from existing systems rather than building software systems from scratch [6-8]. It is well-known that software reuse can improve the productivity and software quality. The reuse process can be conducted at any stage of the software development process [9]. However, in some cases it is not a trivial task to find the appropriate artifices for the reuse purpose. In some others, some scattered fragments can be reused; however the cost of adapting or combining them would be too expensive.

Indeed, reuse of the existing software systems can be seen from different angles: the reuse of the artifacts themselves, or the reuse of the potential knowledge.

1.1.3 Knowledge Reuse

With such difficulties discussed above, it is difficult to utilize the accumulated experiences. Therefore, it is most beneficial to reuse the existing knowledge, in the available pairs (transformation examples), instead of reusing the software fragments themselves. Here the term knowledge reuse refers to the utilization and guidance provided by the past experience, previously developed software systems, to deduce the potential transformation rules and then apply them on the new source instance to come up with the corresponding target models.

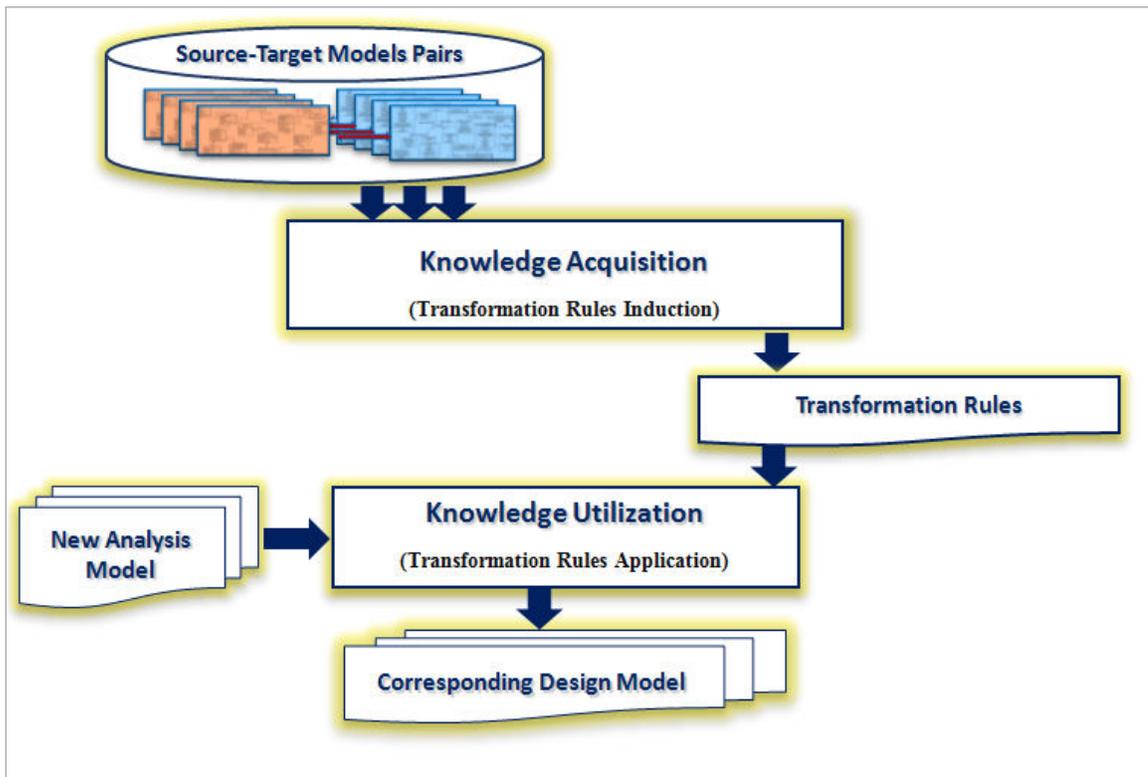


Figure 1: High level view of proposed process

Figure 1 demonstrates that the previously developed software can be utilized and reused to assist the transformation from the source models (defined as the problem space) toward the target design (defined as the solution space) in an automatic manner.

That is, the focus here is to build capabilities that would allow learning transformation rules from available source-target pairs. Rules have been used widely as a powerful way for representing knowledge. The term “*rule*” in artificial intelligence (AI) is used for knowledge representation, and it can be defined as an IF-THEN structure [10]. That is machine learning algorithms can assist in learning this structure of rules based on the available experiences.

In particular, we assume that the transformation rules, induced from knowledge reuse have two characteristics. Firstly; they are declarative rather than procedural. Secondly; they are comprehensible and insightful to a domain expert.

1.2 Motivation and Objectives

As discussed above, model transformation is considered a central part in model driven software development [11]. Several transformation languages have been proposed to perform model transformations. However, to be able to carry out the transformation between different models there is a need to apply the appropriate transformation rules.

Based on that, it is not the whole solution to have a good transformation tool or language that can implement the transformation rules. It is important to have the methodology to define/gather the existing knowledge that describe how the transformation can be achieved [12]. Most often the model transformation approaches rely on rules provided by

the experts [13]. In addition, the available developed software systems, pairs of source-target models, contain the knowledge that can be utilized to specify the needed transformation. This knowledge can be acquired and reused by learning and formalizing relationships linking source model artifacts to their corresponding target model artifacts and then applying the formalized rules on new instances. Nevertheless, it is difficult to discover and formalize valid, potentially useful, and eventually understandable patterns in software models.

In the domain of model transformation, authoring, expressing and maintaining the suitable transformation rules is not a trivial task, particularly for non-widely used formalisms [14]. It would be a very expensive and time-consuming task when constructing a knowledge base for some expert system by interviewing experts and writing down the rules they give. It would be much easier for the human experts to provide transformation examples (pairs of source and target models) rather than provide a complete and consistent set of transformation rules. Thus, it would have been highly beneficial if the transformation rules can be captured automatically, even partially, to utilize the accumulated experience. The generation of the rules can be considered as a search problem guided by the examples, where solutions are searched based on examples. Such learning from examples will be the topic of this dissertation.

An automatic rule capturing, e.g., by applying machine learning (ML) algorithms, can be of help even though the automatically produced rules are not guaranteed to be correct [15]. Thus, practically, a learning approach can be used to deduce the transformation rules from the available set of examples [16]. Accordingly, there is a need for intelligent tools that support the transformation from requirements analysis to software design

utilizing the analysis-design pairs. ML techniques have been applied in different domains, including software engineering [17-19]. ML techniques offer algorithms that have the ability to enhance their performance automatically through experience [19].

ML methods are often used to discover and learn patterns and relations in the available data [20]. Inductive learning can be utilized in this regard to determine decision rules from examples of expert decisions. This type of learning can significantly simplify the transfer of knowledge from an expert into a machine [21].

Inductive logic programming (ILP)[22, 23] which utilizes the power of ML and the capability of logic programming to induce valid hypotheses from a set of examples. ILP is a machine learning technique that provides mechanisms for inducing valid hypotheses from given examples and background knowledge of the domain of interest [22, 24].

It is noticeable; ILP concerned with learning logic programs from examples, which is the same concept of MTBE which aims to learn transformations from examples. ILP is appealing for the problem of learning transformation rules because it can be used to capture both the relational nature of the different models artifacts and their dependencies. Therefore, in this dissertation, we direct our objectives toward employing inductive learning in order to automatically induce transformation rules of MDD based on a set of source-target pairs' examples.

Moreover, with more examples the resultant transformation rules can be exploited to build a rule-based expert system for automating the transformation process of analysis into design.

Transforming the analysis models into software design models can be viewed as a model transformation problem. Thus, the analysis-design models transformation is considered the case study in this work. The provided analysis-design pairs have been utilized in deriving the transformation rules and then applying the rules on the given requirements to produce the corresponding design.

1.3 Dissertation Contributions

The goal of this research effort is to investigate and develop necessary algorithms and tools that utilize the examples and automatically generate transformation rules. The process we follow here is comprised of three different phases. In the first phase, rules induction, our proposed transformation system employs an ILP system to induce the transformation rules using the available source-target pairs' examples. In the second phase, the induced rules are evaluated to find the best and most generic rules. Finally, in the third phase, the top ranked rules are validated against unseen examples. The objective of the evaluation and validation phases is selecting the best groups of the induced rules to be considered in the rules base for the future development.

The proposed transformation system can employ inductive learning agents to induce the transformation rules by utilizing interrelated source- target models. Our approach differs from other proposed model transformation approaches in many facets: it can use the minimal input, concrete source/target models, without meta-models or transformation mappings. Another important contribution is a new ILP system that uses genetic algorithms to induce the rules from the given examples. The contributions of this work can be summarized as follows.

- A comprehensive literature survey on model transformation approaches, especially those proposed to induce transformation rules.
- An ILP-based model transformation system for inducing, evaluating, validating and applying the transformation rules.
- Format-conversion tools to allow using artifacts available in standard formats (XMI¹).
- Evaluation of one of the common ILP systems, namely ALEPH, to induce MDD transformation rules.
- A new ILP system using genetic algorithms for generating rules based on the examples.

The proposed transformation system was validated in the context of model driven development transformations. Mainly we consider one of the modeling views, class diagram, however in some examples we use another view of modeling i.e., sequence diagram. Our literature survey revealed that the problem of automatically moving from analysis to design through utilizing previously developed projects has not caught enough researchers' attention yet [25].

1.4 Dissertation Organization

This dissertation consists of nine chapters, including this introductory chapter. Each chapter is broken down into separate sections and subsections. The rest of the dissertation is structured as follows:

¹ XMI (XML Metadata Interchange) is "an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML)."

- Chapter 2 gives background material on the model transformation, which includes: transformation types, model transformation languages and transformation by examples. Then, the concepts behind inductive learning and definitions of inductive logic programming are given. It describes the basic techniques in ILP which are used in the proposed system. Finally, it presents a brief description of the rule-based expert system and the JESS rule engine.
- Chapter 3 provides the literature review of the state of the art related to this work. The work proposed in this dissertation crosscuts four research areas: (1) transition from analysis to design, (2) transformation approaches in model-driven development; (3) model transformation by examples approaches; and (4) applying inductive logic programming in various domains.
- Chapter 4 discusses the representation of the knowledge. It shows the road map we follow. We review and investigate strengths and weaknesses of using Prolog in software design patterns detection. It introduces also the first order logic predicates used for defining the models artifacts and relationships.
- Chapter 5 starts by listing the research questions we aim to answer by this work. It then details the proposed solution which is an ILP-based transformation system. The proposed transformation system consists of three components in addition to the pre/post-processing phases.
- Chapter 6 presents the experiments setups including the transformation tasks, datasets, and the used performance measures. Then it focuses on the experiments conducted using ALEPH system. Finally, it shows the limitations of ALEPH system in the context of MDD transformations.

- Chapter 7 presents the architecture of the proposed ILP system, called MTILP. It focuses on the concepts behind the proposed system, and the proposed learning algorithms.
- Chapter 8 discusses the experimental results of using MTILP system. Then it compares the results of applying ALEPH and MTILP systems in the context of MDD transformations.
- Chapter 9 concludes this dissertation. Here, a section summarizes the work briefly and another section details the accomplishments of the work performed in this study. Last two sections to discuss the study's limitations and survey possible improvements as opportunities for future work. |

CHAPTER 2

BACKGROUND

This chapter provides further background overview on model transformation, inductive learning and expert systems. The aim of this overview is twofold: to make the thesis self-contained by providing the unfamiliar reader with enough background to understand it, and to put MTBE and our contributions to it into perspective. In this chapter, Section 2.1 introduces model transformation, its different types, tools and methodologies, and more specifically the MTBE approach. Section 2.2 discusses the learning theory and the need for knowledge engineering. The basic concepts, potential advantages, types of ILP are presented in Section 2.3. Finally, Section 2.4 gives an overview of the rule-based expert system and presents the Java Expert System Shell rule engine to be used for executing the transformation rules in this work.

2.1 Model Transformations

In the context of this work, a model represents an abstraction of some real system. A model transformation is a central concept in model driven approaches, where the models are considered the primary artifacts in the software development lifecycle [11].

The concrete syntax of the corresponding system can be represented using graphical notations that help domain experts in constructing understandable problem descriptions [26]. The Unified Modeling Language (UML) is defined as “*a standard graphical notation for modeling any type of system. It has become the de facto standard for*

modeling software systems, adopted and maintained by the Object Management Group (OMG)” [27].

In MDE, the models that represent the system from one aspect can be transformed to other models that describe the same system from another view.

The aim of the transformation process is to generate automatically the target model conforming to a target *meta-model* from the source one that conforms to a source *meta-model*. The meta-model defines the abstract syntax and static semantics of domain-specific modeling languages as shown in Figure 2.

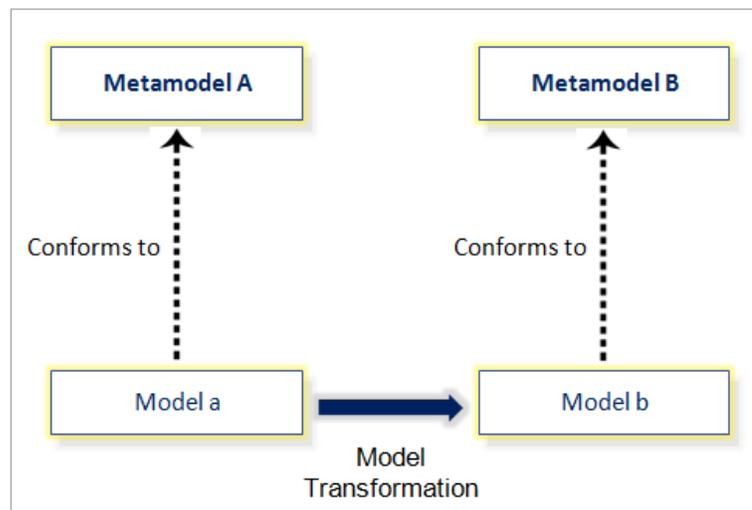


Figure 2: Model transformation process

Several types of transformations can be conducted among the different models. Some transformations need to include the meta-models of both source and target models. Based on that model transformations can be categorized into two main types: *endogenous* and *exogenous* transformations. Figure 3 demonstrates general transformations and some examples of them.

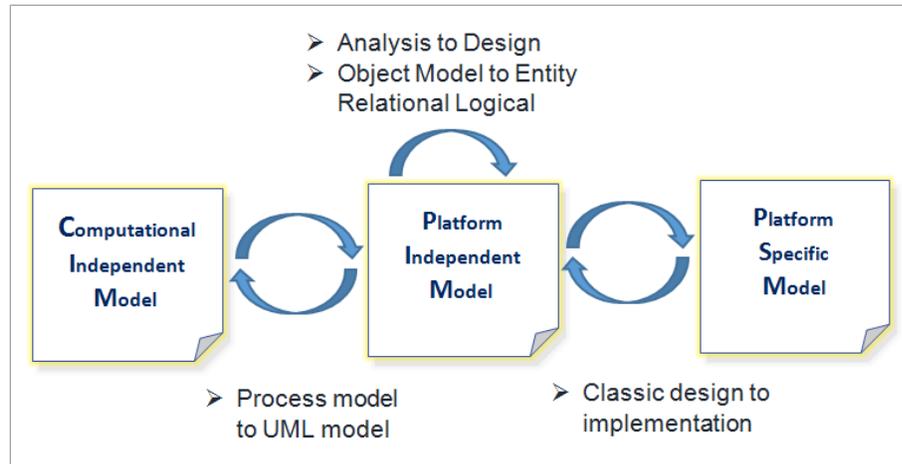


Figure 3: Types of MDE transformations

The endogenous transformation refer to the type of transformation performed between source and target models which are represented using the same language [28]. It rewrites the input model to produce the output model for renaming, adding, detailing or deleting some of its artifacts. An example of endogenous transformations is a transformation of analysis models into design models. This transformation can be achieved during PIM-to-PIM transformations.

The second type, an exogenous transformation, refers to the transformation between source and target models expressed using different languages, i.e., different source and target meta-models. A typical example is the transformation of the UML class diagram to a Java code, known as model-to-code transformation.

In exogenous transformation, the entire source model artifacts must be transformed to their equivalent in the target model. In contrast, in the endogenous transformations we can distinguish two steps: 1) the identification of source model artifacts (only some model fragments) to transform, then 2) the transformation itself. For example, when conducting the transformation from analysis model to design model, the type of

relationships can be identified to be transformed only, while the rest of the artifacts will be presented in the design as they appear in the analysis.

The models can be transformed from one model to another based on a predefined transformation definition which is a set of transformation rules. Basically, each transformation rule describes how one or more artifacts in the source model can be transformed into one or more artifacts in the target model [5]. The transformation rule usually consists of two parts: a *Left-Hand Side* (LHS) and a *Right-Hand Side* (RHS). The former represents the configuration of artifacts in the target models which are subject for creation, update or deletion by the rule. On the other hand, latter represents the configuration of the source models constructs to which the rule applies.

The model transformations, in the literature, have been defined and performed by using various techniques. Some transformation languages have been used by these techniques to define transformation rules and their application.

2.1.1 Modeling Tools and Transformation Languages

Since the introduction of MDE, a large number of model transformation languages have been proposed for model transformation. Each of this language has a unique syntax and style. Indeed, most existing languages and tools provide support to specify and execute exogenous transformations (for stepwise producing implementation code from designs). Nevertheless, endogenous transformations can assist automating many modeling activities to increase productivity of modeling and improve the quality of models.

Model transformation languages have been classified based on various criteria. The reader may consult [11, 13, 28] for taxonomies classifying the model transformation languages in more detail.

Common and traditional approaches toward implementing model transformation rely on specifying the transformation rules, and then employing a model transformation language to automate the transformation process. That is, although the proposed languages have the capability to implement different model transformation tasks, they need to be provided by the corresponding transformation rules. Hence, having a good transformation language is not the solution; it is a part of the solution. The first part is to gather and formalize the existing knowledge to define the transformation rules.

To address the aforementioned limitations and simplify the task of developing a model transformation the user can describe the type of the desired transformation, then a modeling tool induce the transformation rule according to the given examples. Model transformation by examples is an approach that enables this concept [29, 30].

2.1.2 Transformations by Examples

MTBE is a novel approach, based on another by-example approaches such as programming by example [31]. MTBE aims at learning a model transformation from a set of existing examples, where each example represents a pair of the transformation source and target models. From these examples, the used MTBE approach can learn the transformation rules

In the literature, the MTBE approaches derive transformation rules from a set of interrelated source and target models. The input models have to be established by the

user. A matching between models is created to help the learning of rules. Moreover, further source-target model pairs can be added later to refine the derived transformation rules. Most of the MTBE approaches define the initial example as a triple consisting of three elements. The given example is represented by the input model and its equivalent output model, and mappings between the input and output artifacts $E = \langle I, M, O \rangle$. Figure 4 shows the main components of the input example.

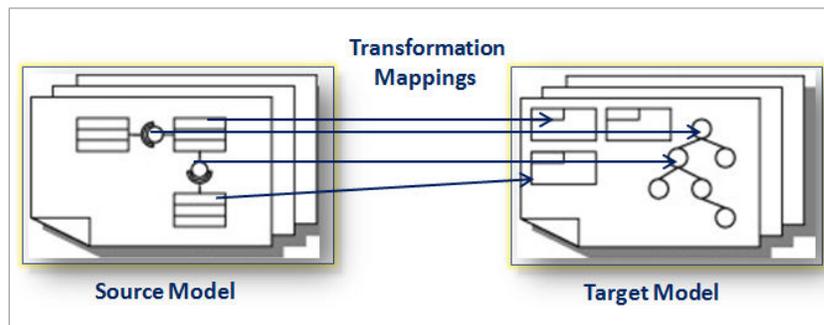


Figure 4: A form of the transformation example

2.1.3 Transformations by Demonstration

Model Transformation by Demonstration (MTBD) [32, 33] is a similar approach to MTBE. It has been proposed for reducing the effort of writing model transformation rules manually. MTBD aims to teach the computer new behaviors by explaining actions or concrete examples.

In MTBD approaches, the end user can demonstrate the desired transformation in the modeling editor. That is, instead of using examples, the user can directly edit the model instances to simulate the model transformation process. The user has the ability to add, delete, update, or connect the model instances to demonstrate how the model

transformation should be done step by step. In other words, by editing the source model and demonstrating the target model changes.

The actions of editing the example models are recorded, then they can be serve as transformation patterns that can be later replayed on other models in future developments.

2.2 Learning Theory

Machine Learning is the branch of Artificial Intelligence that deals with how to devise algorithms that allow automated learning from experience [20]. Learning can refer to any process that can be employed by a particular system to improve its performance [21]. Improving the performance can be achieved through acquiring more knowledge about the surrounding environment or by getting more experience from the produced outcomes. Thus, knowledge acquisition is defined as the capability to learn new information besides the ability to apply that gained information in an effective way.

ML can be divided into two main categories: *unsupervised* and *supervised* learning. In unsupervised learning, the instances are not labeled and the learner seeks to determine how the data is organized. The goal of this type of learning is to let the system learn how to do something without telling it how to do. A common task in unsupervised learning is clustering, where the goal is to organize the given data into a set of distinct clusters whose instances are more similar within a cluster than with instances from other clusters. On the other hand, in supervised learning, the training data consists of instances labeled with the desired target output. The task is to learn a function that generalizes from the supplied examples to unseen ones. This generalization is needed in order to classify

unseen instances. Our work with ILP in this dissertation is always in the supervised learning setting. Hereafter, we will refer to supervised learning simply as learning.

2.2.1 Learning: A Continuous Process

To design a system we need to consider the expected behavior of the intended system. That is the system should know and act upon many requirements, such as the following:

1. *IF there is X1 THEN do Y1 as an action*
2. *IF there is X2 THEN do Y2 as an action*
3.

It is clear that this list can be continued without end. Again and again, one can think of new situations that the system should be able to consider. It seems impossible to take all these requirements into account explicitly in the construction of the intelligent/expert system. The task of coding each of the infinite number of requirements into the intelligent system would be too much. Thus, a solution of this problem is to provide the system initially with the available amount of general knowledge about what it should do, and provide it with the ability to learn from the way the environment reacts to its behavior. In that way, the system does not require to know everything beforehand. Instead, it can build up most of the required knowledge along the way. Thus the system's ability to learn would evaluate and optimize the whole process.

In the work presented in this dissertation we aim to exploit the concept of continuous learning to enrich the model transformation rules in twofold. Firstly, by acquiring the transformation knowledge through learning structural descriptions from a set of positive training examples. The aim is to find a clear and easily understandable hypothesis

explaining the given data. Secondly, by refining and updating the existing transformation rules through exploring more examples.

Inductive learning is a machine learning technique that has been employed widely to discover/acquire knowledge from “real world” data. The next section discusses this learning paradigm in more detail.

2.2.2 Inductive Learning

The terms “Inductive learning” and “learning by examples” are used interchangeably to refer to acquiring knowledge and experience, i.e. learning a general theory, from specific examples. In this learning process, the examples are considered as the main source of scientific knowledge since they can be used, by the learning process, to discover and highlight existing unobvious patterns or to draw new conclusions.

The inductive learning task can be seen as a search for reasonable general patterns. These patterns describe the provided input data (i.e. the training examples represented as facts concepts, etc.) and they can also be utilized to predict new data (testing examples).

Inductive learning programs have substantial applications in several experimental science including chemistry, medicine, genetics, and biology [34]. The inductive programs can be utilized in two basic modes. In the first mode, they are used as interactive tools for knowledge acquisition from explicit examples (facts). In the second mode, the inductive program is used as part of a ML system, where other components provide the required learning examples for the inductive program. Figure 5 demonstrate the utilization of the inductive learning in a predication system.

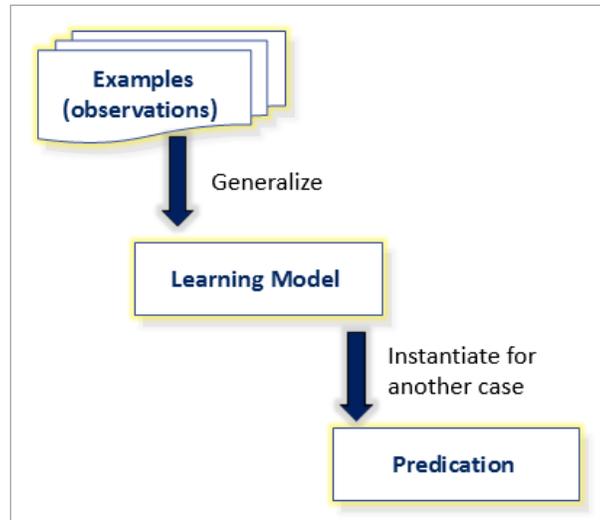


Figure 5: The application of inductive learning

That is, the drawn conclusions (patterns, rules, etc.) rules not only specify something about the given examples, but are in fact about all similar situations. Accordingly, they have predictive power: they can be used to make predictions about future instances. Thus, in the context of learning model transformation rule from the available transformation examples, the induced rules can be used to be applied to generate the target models based on the source models for future development.

2.3 Inductive Logic Programming

When Stephen Muggleton introduced the name "*Inductive Logic Programming*", he defined this field as the intersection of ML and Logic Programming [24]. Thus, it inherits characteristics of both ML and logic programming. As the name suggests, ILP generally uses inductive inference to generate hypotheses from examples (also called observations) presented with the background knowledge. Thus, ILP is appealing for the problem of

transformation by examples because it can be employed to learn the model transformation rules by using the available transformation examples.

This section gives a detailed overview of ILP, the chosen ML technique used in this research. The aims of this chapter are to introduce the basic ILP terminology and its different types.

2.3.1 Basic ILP Concepts

ILP starts the learning process by using an initial background knowledge K and some examples E to induce valid hypotheses (a set of rules) H and to draw new conclusions [24, 25]. The examples, background knowledge and the induced hypothesis in ILP are represented in a declarative manner (as logic programs).

The power of using the background knowledge is that it gives the domain experts the opportunity to select and integrate the appropriate knowledge that describes the given examples and the undertaken domain.

Usually the examples are categorized into positive E^+ and negative E^- examples. The former are consistent with the induced hypotheses and the background theory, while the latter are contrary to both of them. The general ILP learning problem can be defined as follows:

***Given** a background knowledge K containing definitions of predicates and a set of positive E^+ and negative E^- examples where $E = E^+ \cup E^-$.*

***Find** a hypothesis H such that $\forall e^+ \in E^+$ s.t. $K \cup H \models e$ and $\forall e^- \in E^-$ s.t. $K \cup H \not\models e^-$ i.e., the background theory together with the hypothesis entails all positive examples and none of the negative examples.*

That is, ILP attempts to find a rule which will be consistent with the positive and negative examples of given facts (background knowledge). In ILP terminology, a rule covers the set of facts if it is consistent with every positive and negative example. H is a complete w.r.t. the background knowledge and the given examples in case it covers all instances in E^+ set. In another way, H is defined as a consistent w.r.t. the background knowledge and the given examples if it doesn't cover any negative example.

In practice, noise may be presented in some problems. That is, some positive examples are introduced as false while some negative examples are presented as true. Due to this noise, completeness and consistency requirements might be relaxed to solve this problem. In such cases, the target is to build a hypothesis that maximizes the positive examples coverage and attempts to minimize the negative examples coverage [35].

2.3.1.1 Illustrative Example

The example shown in this section illustrates the ILP task on a simple problem of learning family relations. Figure 6 shows the family tree representing the problem to be learned.

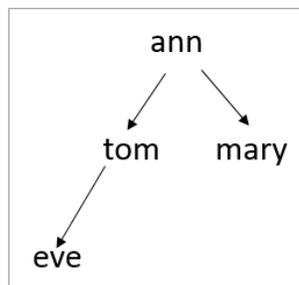


Figure 6: Example of a family tree

To start learning, there is a need to specify the background knowledge, positive and negative examples, and the language bias.

- Background Knowledge

Unlike most ML techniques which only learn from examples of the concept, ILP is able to bias inference by taking into account background knowledge. ILP learners receive the training examples together with potentially relevant background knowledge about the examples. Thus background knowledge is used to feed the ILP learner before the learning starts.

Based on these inputs ILP learner constructs hypotheses that describe the examples in terms of the background knowledge.

The background knowledge describes the facts given in the presented problem. The facts, presented in Figure 7, contain the information and relationships between the people involved in the family tree shown in Figure 6 . This information represented by first order logic predicates.

```
parent(ann,mary).  
parent(ann,tom).  
parent(tom,eve).  
female(ann).  
female(mary).  
female(eve).
```

Figure 7: Representation of the background knowledge

- Examples

In ILP theory, either the positive and negative examples are represented using the predicate relation. Figure 8 demonstrates the positive and negative examples in family tree to find a particular target relation.

<u>E+</u>	<i>daughter(mary,ann).</i> <i>daughter(eve,tom).</i>
<u>E-</u>	<i>daughter(tom,ann).</i> <i>daughter(eve,ann).</i>

Figure 8: Representation of the positive and negative examples

The negative examples are represented in different ways in different ILP systems. Some systems use headless ground unit clauses, while other systems don't use but they put them in a separate file.

- Target Relation

The target relation needs to be defined, in this induction task, is *daughter* (A, B). In terms of the background knowledge relations "female" and "parent", the "daughter" relation states that the person A is a daughter of person B. Using an ILP learner, it is possible to formulate the following definition that satisfies the given target relation:

$$daughter(A, B) \leftarrow female(A), parent(B, A).$$

We can say that, the induced definition is complete and consistent w.r.t. the given examples and background knowledge. Indeed, the predicates involved in the induced hypotheses are determined by the background knowledge and the specified language bias, which specify the definitions of the target predicates. That is, to satisfy the given target relation, based on different specifications, the ILP learner may formulate the following definitions

$daughter(A, B) \leftarrow female(A), father(B, A).$

$daughter(A, B) \leftarrow female(A), mother(B, A).$

The target relation usually is defined using special statements called *modes*. The following section details the usage of *modes declarations*.

2.3.1.2 Mode Declarations

Most of ILP systems use modes declarations to specify the predicates definitions that can be used to build the learned hypothesis. Two types of modes are used to inform the learning system what predicates can be used in the induced hypothesis: *modeh* (head declarations) and *modeb* (body declarations). The former, *modeh*, describes the format of the head of the target hypothesis, while the later (*modeb*) describes the format of the atoms expected to appear in the target hypothesis inspired by the background knowledge predicates. The aim to use mode declarations is to constrain the search space.

Indeed, most ILP systems are restricted to learn definite horn clauses that take only the form $h \leftarrow (p \wedge q \wedge \dots \wedge t)$. This means the target hypothesis can have at most one predicate as its head.

The head of the rule is obtained from the positive examples, while the body is constructed using the atoms presented in the background knowledge. It is important to mention that, not all the information (atoms) provided in the background knowledge needs to be included in the rule body.

In the family tree problem presented, Figure 9 shows how modes can be declared using ALEPH system [36].

```
:- modeh(*, daughter(+person,-person)).  
:- modeb(*, parent(-person, +person)).  
:- modeb(*,female(+person)).
```

Figure 9: Snapshot of modes declarations using ALEPH

2.3.1.3 Bias

As a result of increasing the used number of predicates and arguments, the search space can grow exponentially. Thus, certain restrictions can be used to guide the search of the ILP algorithm. These restrictions usually are called *bias*.

Practically, the intent of using bias is to keep the search space manageable (by reducing the number of candidate hypotheses). However inappropriate bias might prevent the ILP learner from finding the target hypotheses.

Two kinds of bias can be distinguished: *language* and *search* bias. The former specifies the hypothesis space, while the latter restricts the search space of the possible hypotheses.

- *Language bias* can be used to specify constraints on the clauses in the search space. The more restrictions we put on clauses, the smaller the search space, and hence the faster a system will finish its search. However, restrictions on the available clauses may cause many good theories to be ignored. For example, we may restrict the search space to clauses of at most 5 literals, but if only correct theory contains clauses of 6 or more literals, no solution will be found. Thus there is in general a trade-off between the induced hypotheses quality and the ILP-system efficiency. One importance aspect of language bias is the capability of a system to introduce new predicates when needed. A restriction of the language to

the predicates already in use in the background theory and the examples may sometimes be too strict.

- *Search bias* specifies the way the system searches its space, i.e. the available clauses. One of ways is to perform exhaustive search by exploring the whole search space. However, usually execution time is one of the deficiencies of exhaustive search. Thus, it is better to have certain heuristics that guide the search. That is, specifying which parts of the space can be searched and which parts can be ignored. Again, this may cause the system to overlook some good theories. It results in a trade-off between efficiency and the quality of the final theory. Therefore, if a system has found that a correct theory is not available using its present language and search bias, it can try again using a more general language and/or a more thorough search procedure. This is called shifting the bias.

2.3.1.4 Structuring the Hypothesis Space

The purpose of the learning task is to find a hypothesis that satisfies some criteria. Such criteria are typically expressed in terms of coverage of the given data while considering the constraints. Hence, the ILP problem can be viewed as a search problem, where there is a space of candidate solutions to be searched [23]. The learner searches the space of available clauses using a particular search strategy and search heuristics. The learner can search the hypothesis space blindly, in an uninformed way, or heuristically. In uninformed search, the depth-first or breadth-first search strategy can be applied. On the other hand, in heuristic search, the usual search strategies are hill-climbing or beam search.

In the ILP problem context the search space can be determined by the language of logic programs L that consists of a set of clauses. Each clause is represented in the form $H \leftarrow B$, where H is the target predicate $h(arg_1, arg_2 \dots \dots, arg_n)$ and B is a conjunction of atoms $t_1, t_2 \dots \dots, t_n$. The atom t_i , represents the body the clause, is determined by the atoms presented in the provided background knowledge.

Representation of an ILP problem contains different clauses that consist of many atoms and various arguments. To structure this space there is a need to introduce a partial ordering into a set of clauses based on the so-called θ -subsumption. To define θ -subsumption, we first need to define substitution. A substitution $\theta = \{X_1/t_1, X_2/t_2, \dots, X_n/t_n\}$ is a function of variables to terms [37, 38]. Let C_1 and C_2 be two program clauses represented as finite sets of literals. For instance, let C_1 be the clause $t(A,B) \leftarrow q(A)$ and let C_2 be the clause $t(A,B) \leftarrow q(x)$. Now $C_1 \theta - subsumes C_2$ under the substitution $\theta = \{A/x\}$. θ -subsumption imposes a partial ordering between the clauses, which can be represented in the form of a subsumption lattice. In addition, θ -subsumption introduces also the syntactic notion of generality.

We say that C_2 is a specialization (refinement) of C_1 and that C_1 is a generalization of C_2 . There is an important property of θ -subsumption: if $C_1 \theta - subsumes C_2$, then C_1 logically entails C_2 (C_2 is a logical consequent of C_1). It is worth mentioning that, the reverse of this relation is not always true [24]. The θ -subsumption concept provides the foundation for two important ILP techniques, shown in Table 1, generalization and specification [39].

Table 1: Generalization and Specification

Generalization	Specification
Bottom- up direction search in the hypothesis space	Top-down direction search in the hypothesis space
specific-to-general	general – to- specific
Starts the search from the most specific clause covering a given example and then generalizes the clauses repeatedly.	starts from the most general and repeatedly applying specialization to the clauses
If the generalized clause starts covering negative example, then all the generalizations are pruned (due to inconsistency).	If a clause does not cover a positive example, then all the specializations of the clause be pruned

2.3.2 ILP Dimensions

Inductive logic programming systems (as well as other inductive learning systems) can be divided along several dimensions.

2.3.2.1 Batch Learning and Incremental Learning

The distinction between batch learning and incremental learning concerns the way the training examples E are given. In the ILP systems that consider only batch mode, all training examples are given right at the outset and not changed afterwards. This model has an advantage of the ability to measure and deal with the noise (errors in the given examples) by applying statistical methods, if needed, to the set of all examples [39]. Since the treatment of noise is usually application-dependent, we will not give much attention to noisy examples in this dissertation.

In the incremental mode, the training examples can be injected through different runs and the system adjusts its theory based on the examples given each time.

In ILP systems, usually most of the systems have predefined background knowledge, and the systems keep it unchanged during the learning process. However, there are some systems that allow changing the background theory by adding more facts or retracting some others. This property is called *theory revision*. Although modifications of background knowledge are possible, these systems observe the principle to stay most closely to the initial background knowledge and to do minimum changes. Usually systems with theory revision are *incremental multiple predicate learning systems* [40].

2.3.2.2 Interactive Learning and Non- Interactive Learning

In some situations, an ILP system needs to interact with the user for obtaining some extra information to achieve the learning process. ILP systems that allow this kind of interaction are categorized as *interactive systems*. That is, in an interactive ILP, the ILP learner can interact with the user by posing questions or asking for prompt decisions about the anticipated interpretation. Usually the asked questions query about the intentional interpretation of clause or example. For instance, they can ask the user whether some particular ground atom is true or not. Indeed, the obtained answers from the user help pruning large parts of the search space [24, 39].

On the other hand, in the non-interactive ILP, the ILP learner is not allowed to interact with the user to get more information.

2.3.2.3 Single Predicate Learning and Multiple Predicates Learning

In single predicate learning (SPL), all positive and negative examples provided to the system represent only single predicate. The target is to induce a hypothesis for this predicate [41]. However, the induction process may end up with a set of hypotheses, all of them define the same single predicate [40].

Indeed, most of the ILP systems are typically designed for single predicate learning, which means that they usually restrict the example set to induce one single predicate.

Although SPL systems are most popular ILP systems, multiple predicates learners (MPL) algorithms are more powerful. The ILP literature has largely ignored this problem since it is more difficult. However, interest in such systems has grown recently [42].

In MPL, the aim is to learn multiple predicates. Thus, it is supposed to provide different learning sets (corresponding to multiple predicates) in such a way that all predicates can be learned. It is not a trivial task to achieve this due to the possible contradiction. A specific order, in some cases, is needed to determine which predicate can be learnt first.

To achieve MPL there is a need to consider incremental learning and theory revision. This allows interacting with the user to provide more information such as the predicates learning order and adding more information to help the learning decisions.

In the context of model transformation problem, presented in this dissertation, many-to-many transformations rules represent a significant part in the transformation problem. That is, it is needed to have some means of learning the rules presented as multiple predicates. Thus, we will study the problem under the following assumptions: the whole

example set is given initially, a theory revision is allowed, no consultant can be consulted and the initial theory contains no definitions for the predicate to be learned.

2.3.3 ILP Systems

Several ILP systems have been developed. They employ various search strategies. Generally, ILP systems are clustered around two basic induction methods, top-down and bottom-up. FOIL [43], GOLEM [41], PROGOL [44, 45], CIGOL [46] and ALEPH [36] are well-known ILP systems. In the following we present a brief description about each system.

2.3.3.1 Top-Down ILP Systems

Top-down ILP algorithms learn program clauses by searching a space of possible clauses from general to specific.

- **FOIL**

FOIL [43] is considered one of the common top-down ILP systems. It is one of the earliest first-order logic inductive learners. FOIL is a non-incremental and non-interactive system. FOIL is a batch learning system that requires positive and negative examples for learning. If the negative examples are not provided explicitly, FOIL utilizes the Closed World Assumption (CWA[47]) concept to generate the required examples automatically.

In the search process FOIL adopts the covering approach by using refinement graph. It only finds minimally sufficient clauses that can distinguish positive examples from negative ones. The idea of covering algorithm is to be repeated in order to find clauses

that cover the given positive examples. Then the covered positive examples are removed, and the covering process repeated again until covering all given positive examples.

As a target, to prune huge parts of the hypothesis space, FOIL employs hill climbing. FOIL does not evaluate the quality of a hypothesis, but its information gain heuristic. For the induction of a single clause, FOIL starts the search with an empty body. Then it grows a clause by adding a literal one at a time in a greedy manner to the body of the clause. At each step it adds a literal, and evaluates the information gain heuristic value $H(C)$ of clause C as shown in Equ. 2.1.

$$H(C) = -\log_2 \frac{E^+}{E} \quad (2.1)$$

$$G(L) = n * (H(C) - H(C_{new})) \quad (2.2)$$

In Equ. 2.1, E and E^+ refer, respectively, to the number of all examples and positives which have been covered by C . When adding a new literal L to C , there is a need to calculate the $H(C_{new})$ for the produced new clause and evaluate it. Equ. 2.2 is used to find the score obtained when adding L to the clause C , where n refers to the number of positives covered by C and C_{new} . Afterwards, L with the height gain value is selected. Finally, based on the consistency of the new clauses it can consider and the covered positive examples are removed. The previous steps are then repeated for the rest of positive examples.

FOIL uses hill climbing, which may lead to pruning vast parts that contain the next hypothesis. In addition, it stops adding literals to the hypothesis clause when the clauses reach a pre-defined minimum accuracy.

- **PROLOG**

PROGOL [44, 45] uses a top-down approach and employs inverse entailment and falls in the category of the interactive ILP systems. PROGOL considers the covering algorithms and performs a search through the refinement graph. It attempts to find the minimal amount of clauses that best explain pre-classified observations. PROGOL requires a set of mode declarations, provided by the user, as input to prune the hypothesis space. As explained earlier in this chapter the modes are used to specify the atoms to be used in the intended hypotheses. PROGOL considers the mode declarations as an essential input besides the background knowledge and the examples.

PROGOL starts the search with a clause that has a head specified by the *modeh* and no atoms in the body ($p(x) \leftarrow .$). Then it selects a seed example to create the bottom clause (\perp), it is also called most specific clause. The search space is determined by the bottom clause. PROGOL attempts to discover the minimal amount of clauses that best explain the given examples. To measure the quality of the candidate clauses it uses a function f_c (called a compression value) shown in Equ. 2.3. The value is measured for each clause c to see how well it covers the given examples with preference given to shorter clauses.

$$f_c = p_c - (n_c + l_c) \quad (2.3)$$

Where p_c and n_c are the numbers of positive and negative examples, respectively, covered by c , and l_c is the length of clause $c - 1$. To compute f_c properly, PROGOL requires examples to be classified into positive and negative. This implies that PROGOL can only learn one class at a time. To find the clause with maximal compression value f_c , PROGOL applies an A*-like algorithm.

- **ALEPH**

To the best of our knowledge, ALEPH (A Learning Engine for Proposing Hypotheses) [36] is the most widely used ILP system. It is considered as a top-down relational ILP system working based on inverse entailment. ALEPH implements concepts and procedures from a variety of different ILP systems and papers. Thus it tries to integrate ideas from many systems and, with the proper settings, can emulate PROGOL. The main algorithm implemented in ALEPH is the same as PROGOL algorithm. However, ALEPH differs by applying various evaluation functions, refinement and operators search strategies.

ALEPH follows a very simple procedure for inducing a hypothesis H , it starts by selecting an example to be generalized, then it builds the most-specific-clause, \perp , for the example. Afterwards, it employs a heuristic search to find a clause more general than \perp but bounded by it. Finally, it keeps the induced clause and removes the examples covered by the induced clause. The algorithm is repeated if more positive examples exist, otherwise the algorithm terminates and returns the set of clauses that have been found.

2.3.3.2 Bottom-Up ILP Systems

Bottom-up methods search for program clauses by starting with very specific clauses and attempting to generalize them. In logic programs, general clauses may be used to prove specific consequences through resolution theorem proving. Bottom-up induction inverts the resolution process to derive general clauses from specific consequences.

- **GOLEM**

GOLEM [41] is one of the well-known bottom-up ILP systems that use covering methods. It builds a clause by considering the relative least general generalization (RLGG) of random pairs of positive examples. GOLEM uses as input the background knowledge and the examples that consist of ground facts only. In addition, mode declarations and negative examples are required by GOLEM in order to reduce the induced clauses numbers and size. Some predefined restrictions are used by GOLEM. For instance, it avoids searching a large hypothesis space, thus it copes efficiently with large datasets. The *lgg* of clauses C_1 and C_2 is the least general clause which subsumes both C_1 and C_2 . An *lgg* is computed by matching compatible literals of the clauses; wherever the literals have differing structure, the *lgg* contains a variable. To induce the hypothesis, GOLEM starts by selecting random subset of the set of positive examples (e_1, e_2) and constructs their relative least general generalization *rlgg* (e_1, e_2) as shown in Equ. 2.4.

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K)). \quad (2.4)$$

Where e_1 and e_2 refer to two used positives and K refers to the background facts conjunction. GOLEM selects one *rlgg* among all the acquired *rlggs*. The selection process considers two criteria: the largest number of positive examples, and consistency with the presented negative examples. The obtained clause can be further generalized and the covered positive examples are deleted as explained earlier. The covering approach is repeated till it covers all positive examples. At the end of induction, GOLEM conducts a post-processing phase to reduce the induced clauses through removing irrelevant literals.

- **CIGOL**

CIGOL [46] is an interactive bottom-up ILP system which incrementally constructs first-order Horn clauses theories from given examples. CIGOL uses inverse resolution to induce the hypotheses from the given examples. The basic idea is to invert the resolution rule of deductive inference using the generalization operator based on inverse substitution. In a resolution step, two clauses C_1 and C_2 are used to produce the resolvent C . In an inverse resolution step, the resolvent C is taken to produce two clauses C_1 and C_2 . CIGOL uses one of the inverse resolution operators which is the absorption operator. The absorption operator is defined as shown in Equ. 2.5.

$$\text{Infer } "p \leftarrow q, B" \text{ from } "p \leftarrow A, B \wedge q \leftarrow A" \quad (2.5)$$

In the following, Table 2 summarizes the characteristics of the ILP systems presented above.

Table 2: Characteristics of various ILP systems.

System	Top Down	Bottom Up	Non-Incremental	Interactive	Non-Interactive	Multiple Predicates	Predicative
FOIL	✓		✓		✓		✓
PROGOL	✓		✓		✓	✓	✓
ALEPH	✓		✓	✓			✓
GOLEM		✓	✓		✓		✓
CIGOL		✓	✓	✓			✓

2.3.4 Potential Advantages of ILP

Like other techniques, ILP has plus and minus points. However, for the purpose of our research we believe that ILP with its characteristics has potential. It provides a richer representation than other ML techniques. In addition, the results from applying ILP, in the literature, [34, 48, 49] show that using logic has some important advantages over other approaches used in Machine Learning.

- Logic provides a very expressive and uniform means of representation for the input and output i.e., the given examples, the background knowledge and the induced hypotheses.
- Knowledge represented as rules and facts over certain predicates comes much closer to natural language than any of the other approaches. Hence the set of clauses that an ILP-system induces is much easier to interpret us humans than, for instance, a neural network.
- The use of background knowledge which provides more descriptions fits very naturally within a logical approach towards Machine Learning.

In addition, ILP has shown its value through its potential application in the following areas: knowledge acquisition, software engineering, knowledge discovery in databases, inductive program synthesis, inductive data engineering, and scientific application such as drug design and molecular biology. Examples of these studies can be found in Section 3.5. Several potential advantages encourage the application of ILP. In the following, we map the ILP potential advantages to the characteristics of the transformation problem we aim to tackle:

- First, instead of using attributes and logic, ILP uses relationships which can infer new relationships. For example, one encodes that *class A* is in *package P* and the *class A* and *class B* in the same package. The program can infer that *class B* is in the *package P* without need to encode this fact. Based on that, new features can be learnt and deduced instead of encoding them explicitly. Attribute-based approaches, such as neural networks, regression, etc., do not have the ability to learn new attributes.
- Second, ILP provides comprehensible results. That is, the derived rules are presented as logic programs descriptions. These descriptions are both formal and comprehensible at the same time. The choice of having results which are unambiguous and understandable is always preferred. This is because the first-order logic predicates can provide a formal manner of representation that carry only one meaning
- Third, ILP employs a declarative knowledge representation which facilitates the development of background knowledge in cooperation with a domain expert. The software designers would definitely prefer to see transformation rules that are comprehensible such that they can suggest new design ideas and have confidence in these rules. Furthermore, the evaluation of the rules after their application is required to enhance their efficiency.
- Fourth, ILP can provide a powerful mechanism to induce formal relationships among the different artifacts presented in the system models.
- Fifth, ILP is more practical when dealing with data heterogeneity i.e., when the required data are collected from various sources. For examples, the used examples

may represent different UML models such as class diagrams, sequence diagram, etc. In addition, it is necessary to utilize these models as one consistent source file. To use a conventional data-mining algorithm, it is impractical to join all the models in a consistent and integral manner.

- Sixth, ILP system can be used to learn the relationships between analysis and design artifacts. The reason behind that is the ability to construct most specific clause that entails one of the examples presented, and is within language restriction provided.

2.4 Rule-Based Expert Systems

This section introduces an overview about the rule-based expert systems.

2.4.1 An Overview

An expert system is defined as the system that imitates the intellectual activities that make a human an expert in an area such as financial applications.

The key elements of a traditional expert system are a user interface, knowledge base, an interface engine, explanation capability, and a knowledge acquisition system [50]. Figure 10 demonstrates the basic elements of the expert systems.

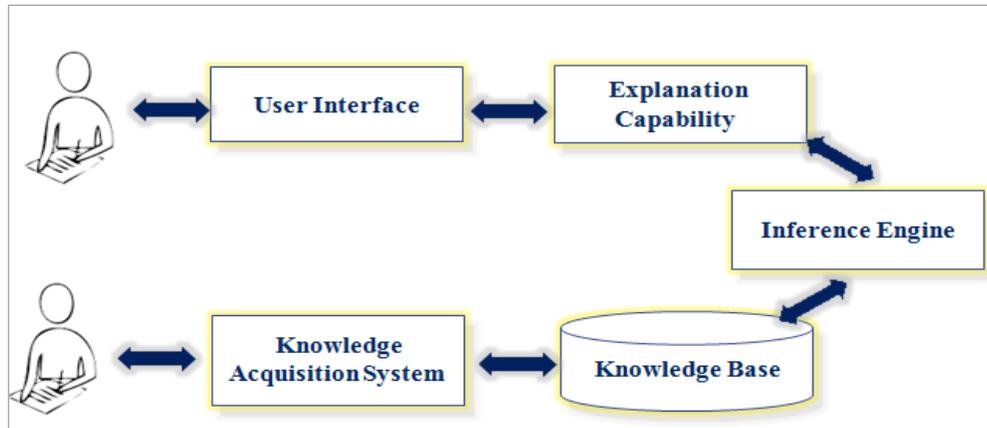


Figure 10: Basic structure of an Expert System

The inference engine uses the knowledge base along with the facts given by the user to draw inferences in making a recommendation. The system can chain the IF-THEN rules together from a set of initial conditions moving to a conclusion. This approach to solve the problem is called *forward chaining*. In a different way, when the conclusion is known and its path is not identified, then *backward chaining* can be used [51].

The derived rules can be written for any rule engine for applying them. In this work, the derived transformation rules are written for the JESS rule engine for applying them. In the next section we introduce the main concepts of the JESS rule engine.

2.4.2 Java Expert System Shell

Java Expert System Shell (JESS) is a rule engine that is integrated in Java platform [52]. Jess supports the development of rule-based systems that can be integrated with code written completely in Java. JESS uses the Rete algorithm for rule inference. It is one of the efficient mechanisms for solving the many-to-many matching problem. In addition, Java code can be referred by JESS code. The software built using Java and JESS has the capability of reasoning using knowledge provided in the declarative rules format.

The JESS program is usually composed of facts and rules. Facts represent the data the rules work with. All the rules define in JESS are stored internally in the JESS rule-base. The facts and rules managed by the rule engine are organized into modules that can be executed at different stages. The rules have two parts: conditions and actions. The former represented by and called left hand side (LHS), while the latter represented by and called right hand side (RHS). When the LHS, matching fact patterns, is (are) satisfied, the RHS, a list of actions, are performed.

$$IF (fact_1 \wedge fact_2 \wedge \dots \wedge fact_n) THEN (action_1 \wedge action_2 \wedge \dots \wedge action_m)$$

Figure 11 demonstrates a very simple JESS rule which displays the name of each person who has a name.

```
(defrule welcome
  ( Person (firstName ?name))
==>
  (printout t "Hello' ?name "!!!" crlf)
)

(deftemplate Person (slot firstname))
```

Figure 11: A sample of JESS rule declaration

The conditions in LHS and facts conform to a *template*. In JESS the template is a similar to the class concept in Java. It defines a fact type. A template has a name and a number of slots. The template instance, i.e. fact, has particular values of the declared slots. The example shown below explains the declaration of *Person* template. The example declares a template named *Person* with a property *firstname*. To instantiate a person fact, we use command *assert*. For instance, (*assert (Person(firstname Peter))*).

CHAPTER 3

LITERATURE SURVEY

In researching a suitable solution to the problem of automatic design development starting from analysis models, we reviewed many approaches in the literature. In this section, we discuss different approaches, and report on a comprehensive list of papers belongs to each approach. The survey addresses four different views (1) transition from analysis to design, (2) transformation approaches in model-driven development; (3) model transformation by examples approaches; and (4) applying inductive logic programming in various domains.

3.1 Transformation from Analysis to Design

In reality it is difficult to move from requirement models to software design automatically. Thus, recognizing the differences between what is modeled in the two phases can help significantly to come up with a more conscious development approach. Kaindl [9] studied analysis and design models of real-world projects. He emphasizes that the transition between analysis and software design is an issue regardless of whether developers use an object-oriented approach or not.

Analyzing the requirements and building the models of analysis and design are cumbersome and complicated tasks which require automated support. Natural Language (NL) is used frequently to describe the software requirements. In a typical software industry, requirements specification is written in NL to enhance communication between

different stakeholders. Due to its inherent ambiguity, it is particularly not an easy task to generate design objects from NL specification. However, structured and constrained NL can be utilized to improve the correctness of the design. Most of the work related to moving from requirements to analysis and then to design only focused on the first transition based on NL processing [53-65]. Some other studies tried to obtain class diagram from use cases [66-68], however the resultant class diagrams are still in the high level description. Similarly, other researchers [69-71] tried to generate other analysis diagrams from use cases.

Therefore, there is a gap when moving from the requirements to architecture. While software architecture is identified formally, software requirements are captured informally. In this regard, a substantial amount of research has been conducted to bridge this semantic gap. For instance, in order to introduce mapping from requirements to architecture, Grunbacher et al. [72] utilize intermediate models which are closer to software architecture. They propose an approach called CBSP (Component, Bus, System, and Property). CBSP has been applied within the context of different definition techniques of architecture and requirements. Liu et al. [73] analyze the gap between software requirements and software architecture to identify the inadequacy of mapping approaches in traditional structured method and object-oriented method. Based on that, they propose a feature-oriented mapping and transformation approach from requirements to software architecture. Kaindl et al. [74] suggest the use of model driven approaches to ease the mapping from the software requirements to the architectural design.

Despite the scientific contributions of the mentioned studies, there is still lack of effective solutions. As shown in earlier work by Kaindl, models of object-oriented analysis cannot

be simply become object-oriented design models. Neither is it possible to transform domain models to design models. There would be the implicit assumption that each and every object class in the domain model would finally end up as several object classes in the detailed design and consequently the implementation. Larmen also states in [75] that domain models represent real-world concepts and not software objects and thus cannot be transformed automatically to a software design, but having mappings between domain and design classes lowers the representational gap between our mental model and the software. Even though automatic transformation seems not possible without intelligent problem solvers, establishing traces like in [76] and mappings between a domain and design model is useful for validation and case-based reuse. The mappings from requirements to design may be viewed as special and elaborate forms of traceability links. All of these concerns motivated Kaindl and Falb [77] to ask “Can the transition from requirements to software design be a model-driven transformation or just a mapping?”. Based on that, they further discussed whether model-driven transformations are appropriate and applicable for moving from requirements to software design.

3.2 Model Driven Development

In the model-driven paradigm, there types of models can be distinguished. CIM (Computation Independent Model) which focuses on the domain a higher level of abstraction. CIM is then transformed to PIM (Platform Independent Model) which is designed without considering the underlying platform. PIM is further transformed into PSM (Platform Specific Model) which includes the technical considerations and the underlying platform [4].

3.2.1 CIM-to-PIM

The transformation from CIM to PIM has not a lot of attention of the researchers. Reviewing and analyzing the existing approaches might give an idea about automatically moving from CIM to PIM if possible. Briefly in the following we listed the evaluation criteria we used to compare the different CIM to PIM transformation approaches.

To compare the existing CIM-to-PIM transformation approaches, we used a framework consisting of eight attributes to accommodate all the different aspects that can be covered by each approach. Table 3 summarizes a review and comparison between these approaches based on the following evaluation criteria:

- CIM covered aspects: CIM consists of two aspects or types of models: a) the business process model (BPM) which is used to demonstrate all the business activities; b) the requirement model depicts the system supporting the business activities.
- CIM representation: this criterion specifies the models used to represent the CIM. From the literature, UML, BPM notations (BPMN), and Data Flow Diagram (DFD) are used to represent the business process model. On the other hand, use case model and feature model are used to describe the requirements model.
- PIM covered aspects: in general PIM describes the design of the intended system; in this regard design may cover the functional, structural, and behavioral perspectives.
- PIM representation: various types of models could be used to represent the PIM such as use case, activity diagram, sequence diagram, and class diagram.

- Transformation mechanism: specifies the transformation method between the source model (CIM) and the target model (PIM).
- Automation: to which extent the proposed approach was automated? Can the transformation be fully automatically accomplished?

Accordingly, we conclude that the CIM to PIM transformation is different from the problem we aim to tackle here. The transformation from CIM to PIM is an attempt to match high-level models related to the business view and other models related to the information system view. Model Driven Architecture (MDA) does not show adequate facility to support the CIM to PIM transformation [78]. Moreover, the Object Management Group (OMG) does not refer to this type of transformations, which hinders the domain experts and business analysts [79]. Such transformation depends on the creativity and experience of the designer, so the quality of the resultant models cannot be controlled.

Table 3: Summary of CIM to PIM transformation approaches

Study	Evaluation Criteria					
	CIM aspects	CIM Represent.	PIM aspects	PIM Represent.	Transformation Mechanism	Automation
Zhang et al. [80]	Requirement model	Future model	Structural	Software Architecture	Responsibilities	Semi
Kardoš et al. [79]	Business process model	DFD	Functional, Behavioral, and Structural	Use cases, Activity diagram, Class diagram , and sequence diagram	Manual	Manual
Kerraf et al. [81]	Business process model, and Requirement model	Activity diagram	Structural	Class diagram	Manual	Manual
Cao et al. [82]	Requirement model	Future model	Structural	Software Architecture	Patterns	Semi
Castro et al. [83]	Business process model, and Requirement model	e3 value model and BPMN	Functional, and Behavioral	UC, Service process	Meta-models Mappings	Partially using ATL
Rodríguez et al. [84]	Business process model	BPMN	Functional	Use cases	QVT and refinements rules, checklists	Semi
Raj et al. [85]	Business process model	SBVR	Structural, and Behavioral	Activity diagram, Class diagram , and sequence diagram	Manual	Manual
Suarez et al. [86]	Business process model	Activity diagram	Structural	Class diagram	Manual	Manual

3.2.2 PIM-to-PSM

Since PIM reflects the features of the problem domain, the model is transformed into PSM in order to implement the PIM. That is, to consider implementation issues and the undertaken platform [5]. PSM may contain features that are presented in PIM, thus PSM is not necessarily a refinement of the PIM [87]. The platform-specific details need to be generated using different tools in order to automate the generation of those details [11].

In this regard, many PIM to PSM transformations studies have been described in the literature. The model-to-model transformation approaches can be categorized into: operational and declarative. The first category is based on rules that explicitly identify how to create the target models elements starting from the source model elements. The second category gives an explanation of the mappings between the source and target models focusing on the relation hold between two models.

Informative surveys of model transformation languages can be found in [13, 28, 88, 89]. Czarnecki et al. [13] classify hierarchically the specification of model transformation approaches based on feature diagrams into a number of classes. The feature model offers a terminology used to describe the model transformation approaches as well as to make the different design choices for such approaches explicit. Mens et al. [88] provide a multi-dimensional taxonomy of model transformations. The introduced taxonomy is more targeted towards formalisms and techniques that support model transformation activities. The taxonomy objective is to position the different model transformation techniques and tools within the appropriate domain. In addition, they aim at evaluating these technologies in the context of a particular activity of model transformation.

A conclusion that can be drawn from surveying the existing rule-based and pattern-based transformation approaches is that they are often based on rules that were obtained empirically. When identifying the transformation rules and automating the transformation process, the common approach followed using a model transformation language such as ATLAS transformation language (ATL [90]), QVT [91], Tefkat [92], etc. These languages still suffer some limitations, despite their ability to implement complicated and large-scale transformations. All the model transformation languages demand a considerable effort on the designers' part to define the intended transformation. This might not be straightforward especially for designers who are unfamiliar with that language.

Moreover, usually the rules are defined at the meta-model level. This demands a deep and obvious comprehension of the abstract syntax of the source and target models. The semantic interrelationships between these models also need to be known. Therefore, it might be a difficult task to define, express, and maintain the transformation rules, particularly for non-widely used formalisms. Another dimension of difficulty may appear when the declarative expressions are not at the appropriate level of abstraction for the end user. This may increase training cost and affect negatively the learning curve.

In some situations, it is difficult to expose the domain concepts because they might be hidden in the meta-model. These implicit concerns show that identifying the transformation rules is a demanding task.

Accordingly, some domain experts may encounter difficulties when trying building model transformations for the domain in which they have extensive experience. This is

because of the difficulty encountered when specifying transformation rules at the meta-model level.

To sum up, surveying the transformation infrastructure, it is obvious that tremendous effort has been made to define transformation languages. These languages can be used to implement the transformation programs through expressing and applying the transformation rules. However, it might be a difficult task to define, express, and maintain the transformation rules, particularly for non-widely used formalisms [93]. That is, having a good transformation language is not the solution. It can be seen as part of the proposed solution. It is important to define/gather the knowledge describing how to perform a transformation of one model into another model for the same system [94].

3.3 Towards Model Transformation Generation

Several model transformations have been identified and a number of techniques and tools have been developed to automate their generation and put them into practice. The context of our approach is model transformation by example. The user has to create model transformation examples. An example consists of a source model and its corresponding model in the target language. Then several techniques can be used, such as relational concept analysis or inductive logic, to derive model transformation rules from the examples. These rules are abstract and not operational. They represent fragments of knowledge and must be arranged carefully to perform the actual transformation.

To address these challenges, several approaches are proposed to assist the specification and the design of model transformation.

3.3.1 Model Transformation by Example Approaches

As discussed in the previous section, implementing MT necessitates two different skills: MDD skills and domain-specific skills. The MDD skills include understanding the meta-models and the transformation environment, while the domain-specific skills require good knowledge about the specification of the transformation including the source, target, and the transformation rules. It is noticeable that the first type of skills is specific to the MDD experts, while the second one is possessed by the domain experts. In reality, the domain experts give transformation examples more easily than consistent and complete transformation rules [14]. In this context, MTBE was introduced to utilize the role of domain experts by providing an initial set of examples.

MDD aims to use platform independent modeling techniques for the sake of abstraction from the software implementation level. On the other hand, the aim of by-example approaches is to simplify the development of systems. Instead of the direct developing, it is possible to utilize the existing examples to draw a clear map. Again, in MDD different transformation scenarios occur between the various models, thus different by-example approaches can be employed for these transformations. With focusing on the transformation by-example idea, it seems more worthy and promising to utilize this approach for the transition from analysis to design based on the available analysis-design pairs.

MTBE utilizes a set of examples that represent past transformations (pairs of source and target models). In addition, in many MTBE approaches it is needed to provide extra information that demonstrates the mappings among the artifacts in the source models and

their corresponding artifacts in the target model. The given input can be to produce transformation rules, or mappings leading to transformation rules.

It is noteworthy that the terms transformation links, transformation mappings and transformation traces have been used interchangeably in the literature to refer to the links between the artifacts in the source model and their corresponding artifacts in the target model. Hereafter, we will refer to such links as transformation mappings.

The MTBE approach has been initiated by Varró [29], who derived the transformation rules from an initial set of examples that includes interrelated source and target models. It is supposed to provide to the system the examples and the transformation mappings along with the respective meta-models. Then the transformation mappings are used to derive the rules which were refined by the developers as a final stage in their approach. It can be seen as a semi-automatic approach that learns the rules using the provided examples and mappings between source and target models. In addition to the source and target models, there is a need to provide transformation mappings linking the source and target models and the meta-models corresponding to the source and target models. This approach derived transformation rules of type one-to-one. When knowing that most of the important transformations need rules of type many-to-many, this can be considered a significant limitation.

Balogh and Varró [95] extended the MTBE proposal was by using inductive programming language [22, 23] in order to automate the transformation process instead of the original ad-hoc heuristic. ILP is one of the machine learning techniques that drives a logic program from the existing knowledge (pairs of source and target models), positive

examples (linked elements of the source and target models), and negative examples (not linked source and target elements). Using this type of knowledge, ILP engine infers a hypothesis for each transformation rule. ILP derives a logic program from the existing knowledge (of source-target pair models), positive examples (linked elements of the source and target models), and negative examples (not linked source and target elements). Using this type of knowledge, ILP engine infers a hypothesis for each transformation rule. Nevertheless, like to the original idea the transformation mappings and meta-models are needed as input by the approach.

Wimmer et al. [30] propose a similar work to the initial one proposed by Varró [29]. However, it differs from Varró's one in that it produces executable ATL [90] transformation rules from existing examples. They define semantic correspondences (mappings) between concrete domain models presented in their concrete notation (modeling languages notations) that allow the derivation of model transformation rules. In addition, the user can reason about the mappings in a notation and with concepts the user is familiar with. It is similar to the derivation method presented in [29]. The proposed approach uses the transformation examples and their mappings to derive one-to-one transformation rules.

The previous approach proposed by Wimmer was later extended by Strommer et al. [96]. The derivation process relies on pattern matching and enables two-to-one transformation rules. This approaches share a common comment with the previous approaches, they were not validated with realistic examples. Strommer et al. [96] approach differs from the approach proposed by Balogh et al. [95] in that the former employs inductive learning while the latter uses pattern matching.

Dolques et al. [97, 98] used and extended the anchor-Prompt approach [99] in order to support the discovery of the transformation link. In [98], they utilized alignment techniques and text analysis to partially generate those links, which helps the transformation designers when designing the examples for MTBE process. Relational Concept Analysis (RCA [100]) has been used in [97] to derive commonalities between the source and target meta-models and transformation links. Compared to Varró approach [95], this approach does not use annotations on transformation links and proposes a set of transformation patterns organized in a lattice. However the transformation patterns cannot be executed directly.

Compared to the ILP-based proposal, the RCA-based approach does not use annotations on transformation links and proposes a set of transformation patterns organized in a lattice.

Saada et al. [101] extended the approach presented in [97] to learn transformation patterns from the examples, then those patterns are analyzed, filtered and transformed into operational transformation rules. These rules can be executed by the Jess rule engine [52, 102] which is fact-based rule language. The Jess program is composed of facts and rules, also its engine is integrated in the Java platform.

Saada et al. [103] investigate and compare two strategies of learning rules with RCA. In the first strategy, for learning transformation patterns they used the provided examples individually, and then they gathered all the derived patterns as second step. In the second, they put all the examples in one go to be used for learning the transformation patterns. In both strategies, the derived patterns are then applied to examples for validation. The

results show that, the first strategy produces patterns that are simple to analyze and proper to their exact examples. On the other hand, the pattern lattices form the second strategy, although they are harder and larger to analyze, they have more complete and specific transformation patterns.

All the discussed approaches suffer some limitations where they resolve the problem of rule derivation partially. They consider only one artifact in the source model to produce one artifact in the target model.

Another MTBE approach that generates many-to-many rules was proposed by Faunes et al. [12]. They claimed that the proposed approach does not require detailed mappings between the models to generate the rules. Genetic programming [104] was adapted for this problem of MTBE. The rules are generated and improved through an iterative process, and then the derived rules are translated into Jess and executed by its rule engine. García-Magariño et al. [105] propose MTBE approach to generate many-to-many transformation rules. The rules are generated from meta-models to satisfy some developer constraints. This approach was implemented with ATL. In addition, ECore [106] has been utilized as a language to define the models and meta-models, while Object Constraints Language (OCL [107]) was used for the constraints. Table 4 summarizes the rules generation approaches discussed above.

3.3.2 Model Transformation by Demonstration

Model Transformation by Demonstration (MTBD) [32, 33] is a similar approach to MTBE. In MTBD the model transformation is demonstrated in the modeling editor. The example models are modified. Then the resultant modifications are recorded. The

general transformation is derived from the concrete changes, and then it may be replayed on other models.

MTBD approaches have been proposed for reducing the effort of writing model transformation rules manually. In MTBD approaches, the end user can demonstrate the desired transformation. This can be achieved by editing the source model and demonstrating the target model changes. Approaches of MTBD only for in-place transformations were proposed by Sun et al. [33, 108] and Brosch et al.[109].

However, this approach requires a high level of user intervention. The difference between the two cited works is that Sun et al. use the recorded fragments directly, however Langer et al. use differencing engine to generate ATL rules. In addition, the approach of Sun is applied to endogenous transformations while the approach of Langer [32] is applied to both endogenous and exogenous transformations.

3.3.3 Analogy-based MTBE

Another track in MTBE area is conducted using the analogy as introduced in [14, 93, 110, 111]. In contrast to the aforementioned approaches, the target of the analogy-based MTBE is not to generate transformation rules. This could be considered as a limitation if the goal is to infer reusable knowledge about transformations.

The approach of Kessentini focuses on using search-based optimization techniques to directly generate the target model from the source model without the rules generation. This could be considered as a limitation if the goal is to infer reusable knowledge about transformations. The idea of analogy here is decomposing the given examples into transformation blocks. The mapping block is viewed as the minimal fragment of source

model aligned to fragment of target model. To transform a new source model, its constructs are compared to those in the example source fragments to select the highly matched ones. The blocks corresponding to the selected fragments may come from different examples, thus they are composed to come up with the “best transform”.

The transformation is viewed as an optimization problem where all the available possibilities are evaluated. Because of the expansion of the search space, the selection and composition of the fragments are carried out through meta-heuristic algorithm including particle swarm optimization [112] and simulated annealing [113]. The analogy-based MTBE approaches do not produce rules. They illustrate and evaluate their proposed approach by transforming UML class diagrams (CD) to relational schemas (RS) by utilizing the available pairs of class diagram and entity relation model. The CD to RS transformation is well-known and the mapping here is not complex. In addition, they applied this approach on more complex transformation: sequence diagram to colored Petri-nets [110].

To sum up, the generation of operational rules from the existing examples can be preferable since it allows those rules to be executed on other source models to directly obtain the target models. In Table 4 most of them are specific to exogenous transformation and use matching techniques to derive transformation rules. The approach we propose is based on the work of inductive logic programming. To execute the generated rules we use the rule engine Jess to facilitate their manipulation and execution.

To sum up, different MTBE approaches have been proposed in the literature. However, none of the proposed approaches tried to tackle the problem of transformations of

analysis into design models. Moreover, none of the proposed approaches considered reusing designers' expertise manifested in previous design effort in proposing design options to given software requirements. In this work, we target building a software design-support system by using ILP to induce transformation rules from available requirement/design pairs. The idea is to use existing knowledge (manifested in the given examples) to automatically derive a set of model transformation rules. Our proposed transformation system does not require detailed transformation traces, but only a set of example models pairs. The proposed transformation system consists of three main components: transformation rules generation and generalization, rules application and evaluation, and rules validation and refinement. Most of the approaches mentioned above do not considered the evaluation and refinements of the induced rules.

Table 4: Generating Model Transformation Rules from Examples

Study	Evaluation Criteria								
	Used Method	Rules Generation	Rules Type	Mapping Type	Inputs	Domain	Evaluation Method	Validation	Automated
Varró [29]	Ad-hoc algorithm	Y	Abstract	One-to-one	Metamodels +Mappings + Concrete Models	CD→ER	-	N	Semi-automated
Balogh and Varró [95]	ILP	Y	Abstract	One-to-one	Metamodels +Mappings + Concrete Models	CD→ ER	-	N	Semi-automated
Wimmer et al. [30]	Manually	Y	ATL rules	One-to-one	Metamodels +Mappings +Concrete Models	CD→ ER	-	N	Manual
García-Magariño et al. [105]	Graph Transformation	Y	ATL rules	Many-to-many	Metamodels + Mappings	UC→Agent Task→CodeCompent	-	N	Semi-automated
Dolques et al. [97]	Formal Concept Analysis	N	Patterns	-	Mappings + Concrete Models	LATEX → HTML	-	N	Semi-automated
Dolques et al. [98]	Anchor Prompt	N	Patterns	-	Mappings + Concrete Models	Code→Tables Ecore→Class	Precision & Recall	Y	Semi-automated
Saada et al. [101]	Relational Concept Analysis	Y	Jess Rule Format	Many-to-many	-Metamodels Concrete Models	CD→ ER	Precision & Recall	Y	Automated
Faunes et al. [12, 94]	Genetic Prog.	Y	Jess Rule Format	Many-to-many	Concrete Models	CD→ ER	Matching Metrics	Y	Automated
Kessentini et al. [14]	Heuristic Search	N	-	-	Concrete Models	CD→ ER	Correctness Metrics	Y	Semi-automated
Our Approach	ILP	Y	Definite Clauses, Jess Rule Format	Many-to-many	Concrete Models	Analysis CD → Design CD	-Accuracy Measures - Expert Opinions	Y	Automated

3.4 Using Prolog Rules to Detect Software Design Patterns

In the literature, some detection approaches use logic based formalisms to encode pattern constraints and inference engines to detect them. Some of these approaches are discussed in the sequel.

Kramer and Prechelt [114] provide one of the first implementations to automatically recognize design patterns using Pat system. The Pat system detects structural design patterns, where rules are used to represent the patterns and prolog queries to execute the search. The design constructions are also represented in Prolog as facts.

Kramer and Prechelt focus only on one type of design patterns i.e. the structural design patterns. For each pattern they only consider a single implementation variant [115]. Their evaluation shows that the precision of design pattern detection, based on four benchmarks, is between 14 and 50 percent which is not sufficient. Although the recall reaches 100 percent the result may still have false negatives².

Similarly, Wuyts [116] uses a logic meta language, called SOUL, to express, reason about, and extracts a system's structure. The program entities and the hierarchy structure of the system are described as facts, where they were extracted using static analysis. The design motifs are described as Prolog predicates on the facts. However, due to the use of a Prolog engine this approach has performance issues. Specifically, it cannot deal with variants automatically, also the inherent complexity when identifying subsets of classes matching design motif descriptions.

² We mean by false negatives the set of design patterns occurrence that were not detected by a pattern detector but they are contained in the design

Static and dynamic analyses were combined by Heuzeroth et al. [117] to detect design patterns. To allow automatic generation of static and dynamic analyses, they provide two Prolog-based languages to specify design patterns. They use SanD-Prolog which is a low-level language that includes Prolog predicates to depict the static structure and dynamic behavior of design patterns. A high-level language, called SanD, was used to specify design patterns. Despite of its effectiveness, this approach suffers some limitations: the specifications of SanD-Prolog seem to be lengthy and complex. SanD tends to be more intuitive, however less powerful.

Huang et al. [118] presented an approach to support pattern recovery. Their approach is based on the structural and run time behavioral analysis. Different types of design patterns are considered; structural, creational and behavioral patterns. The structural and behavioral aspects have been utilized. To depict this representation, they introduce a combination of predicate logic and Allen's interval-based temporal logic. Then to support pattern recovery, the formal specifications could be easily converted into Prolog representations.

Stoianov and Sora [119] propose a logic-based detection approach used to detect a set of patterns and antipatterns. The approach defines Prolog predicates to describe the structural and behavioral aspects of design patterns and antipatterns. Prolog predicates simplify the description of patterns and antipatterns characterized by structural and complex behavioral aspects. They compared the results of the proposed approach with the results provided by a common tool on the same analyzed systems. In the case of two patterns out of five there are significant differences between the obtained results. Table 5 summarizes the studies presented in this section.

Table 5: Summary of logic-based approaches for design patterns detection

Study	Evaluation Criteria				
	Structural	Behavioral	Pattern Representation	System Representation	Matching Technique
Kramer et al. [114]	✓	–	Prolog	Prolog	Exact
Wuyts [116]	✓	–	Prolog	Prolog	Exact
Heuzeroth et al. [117]	✓	✓	Prolog-based language	Prolog	Exact
Huang et al. [118]	✓	✓	Prolog	–	Exact
Stoianov et al. [119]	✓	✓	Prolog	Prolog	Exact

3.5 Existing ILP Applications

ILP has been widely utilized for discovery of concept and classification in data mining algorithms. In concept discovery, the idea is to induce rules based on the existing data. For classification, general rules are generated based on the given data and then they are used for grouping the unseen data.

In reality, ILP has been successfully applied to a wide range of real-world problems in different domains since it is concerned with the induction of logic theories from examples[34, 120]. These areas include, but are not limited to:

- Biomedical area, for example: [121] for gene–disease relationship extraction, [122] explores the applicability of ILP method in the pancreatic cancer disease and shows the accumulated clinical laboratory data can be used to predict the

disease characteristics, and [123] employs ILP to extract associations from pathology excerpts.

- Chemical Engineering: Learning drug structure activity rules. In this scenario the aim is to understand the relationship between chemical structure and activity [124]. ILP systems are used to learn rules relating the structure to an activity and can be used to find promising chemical structures, i.e., to increase the success rate of pharmaceutical companies.
- Biochemical field: ILP was employed to build a model of hexose-binding sites [125], another work [126] used an ILP system, ProGolem, and demonstrated its performance on learning features of hexose-protein interactions.
- Bioinformatics field: ILP methodology was used to extract knowledge from a multi-relational database about deleterious/neutral mutations [127]. ILP induced classification rules that can help to improve understanding of the relationships between physico-chemical and evolutionary features and deleterious mutations.
- Logic rules production [128] to identify a driver's cognitive state in real driving situations to determine whether a driver will be ready to select a suitable operation and recommended service in the next generation car navigation systems.
- Natural Language Processing provides several application areas for inductive reasoning: one application area is part of speech tagging. Inductive reasoning has been applied for several languages, including English [129]. ILP has also been used to extract relations from text [130], which is useful for building knowledge bases from text corpora.

- Text mining field: ILP was applied to discover novel or previously-unknown knowledge from two complementary but disjoint sets of literatures of [131].
- Relevant information extraction [132] to allow inducing symbolic predicates, from text and web pages, expressed in Horn clausal logic that subsume information extraction rules.
- Detection of traffic problems [133] to identify problematic areas with respect to traffic jams and accidents. Those tools could identify most critical sections correctly and learn plausible rules.
- Robot discovery environment [134] to find out to which extent robots can learn about their environment by conducting experiments and collecting data.
- Extracting of rules from blogs and micro-blogs for recommending blogs or web pages to the bloggers [135, 136].
- Learning semantic models and grammar rules of building parts [137].
- Deduction of classification rules in the geographic information area [138].
- Induction of rules for extracting class instances from textual data [139].
- Induction of inclusion axioms in fuzzy description logic [140].
- Exploring the automatic learning of several of temporal relations from data [47].
- Extraction of knowledge about human monogenic diseases [141].

Although ILP has been used in model transformation [95, 142], the proposed approaches tried to tackle problems that are different in their nature from the problem we aim to tackle in this work. Essaidi et al.'s approach [142] was in the context of Model-Driven Data Warehouse, while Varró and Balogh [95, 143] approach addressed transformation from UML class diagrams into relational database. In addition, ILP was used in software

engineering field for learning interface specification. Sankaranarayanan et al. [144] propose a methodology for learning interface specification using ILP. An inductive learner was employed to obtain specifications expressed in Datalog/Prolog.

In conclusion, the conducted survey revealed that none of the current approaches address the problem of transforming requirement analysis models into software design models. In contrast with these approaches, our transformation approach addresses this specific problem. Moreover, to derive the transformation rules most of MTBE approaches that require the source, target models and their meta-models as well as the detailed mapping between these models. Unlike these MTBE approaches, our approach aims to use the minimal inputs, the source and target models only, to derive the transformation rules. Moreover, to tackle this type of transformation the system should generate many-to-many transformation rules that investigate the source model looking for a couple of artifacts and instantiate more than one artifact in the target model.

CHAPTER 4

KNOWLEDGE REPRESENTATION

This chapter addresses two interrelated topics: knowledge representation and knowledge acquisition. In this chapter we present the power of logic programming in knowledge representation.

4.1 Knowledge Representation: An overview

Knowledge representation (KR) is one of the most important subareas of artificial intelligence. When designing a program (or any entity) that is capable of behaving intelligently in a specific environment, there is a need to provide this program with adequate knowledge about this environment.

Knowledge representation is defined as “*the field of study concerned with using formal symbols to represent a collection of propositions believed by some putative agent*” [145].

Thus, the goal of *Knowledge representation* is to employ an appropriate language that capture knowledge about environments, their entities and their behaviors, and also allow inferencing from knowledge. Representation and reasoning are intertwined. In other words, the power of knowledge representation is reflected by the ability to make inferences, i.e. inducing new conclusions from existing facts.

In the context of the underlying problem (MDD Transformations), we are concerned with two uses of formal symbols: to create ontologies (sets of interrelated concepts or terms describing a domain) and another use to represent rules (describing the syntax governing the relations between the terms).

The objective in knowledge representation is to develop a minimum syntax that is formal and unambiguous [146]. In the context of the underlying problem (MDD Transformations), the minimum syntax is designed by capturing the name, type, and relations of each construct in the used models.

The knowledge base consists of facts, rules, and heuristic knowledge supplied by an expert who may be assisted in this task by a knowledge engineer. Knowledge representation formalizes and organizes the knowledge using IF-THEN production rules [51]. Other representations of knowledge, such as frames, may be used.

In general, a knowledge representation language/system can be evaluated through four properties: representational adequacy, inferential adequacy, inferential efficiency, and acquisitional efficiency. In light of these properties, in the section 4.3 we investigate the suitability of using Prolog to represent and detect the design patterns occurrences. In [147] we discussed the power of Prolog programming language in more details.

4.2 Software Artifacts: Formal Representation

In the context of the model transformation problem, we need to define the software models constructs/artifacts formally to be able to discover the existing knowledge in the available examples. In this regards, we define three groups of formal predicate

specifications. The model constructs are defined by the first group of predicates. The second group is used to define the relationships linking the different constructs.

4.2.1 Models Constructs Predicates

The predicates in this group identify existence of model constructs. They focus primarily on two parts: i) the type of a particular construct i.e., package, class, interface, method, and attribute; and ii) whether a particular construct includes other constructs for example, package contains different classes, class has a specific operation etc. Table 6 shows the predicates used to identify the existence and the type of a particular artifact.

Table 6: Predicate to define the artifacts types

Predicate	Meaning
<i>package(P)</i>	is used to define P as a package.
<i>class(C)</i>	is used to define C as class.
<i>interface(I)</i>	is used to define the existence of the construct I as an interface.
<i>method(M)</i>	is used to define the construct M as a type of method.
<i>attribute(A)</i>	is used to define the construct A as a type of attribute

Table 7 demonstrates the predicate used to identify that a particular construct is s apart of another one.

Table 7: Predicate to describe artifacts that have other artifacts

Predicate	Meaning
<i>packageHasClass(P,C)</i>	shows that class C is located inside package P
<i>packageHasInterface(P,I)</i>	shows that package P has an interface I
<i>packageHasInterfaceFacade(P, F)</i>	shows that package P has a façade F
<i>classHasMethod(C, M)</i>	means that class C has a method M.
<i>classHasAttribute (C, A)</i>	shows that class C has an attribute A.
<i>interfaceHasMethod(I, M)</i>	shows that an interface I has a method M.

4.2.2 Relationship Predicate

The predicates listed below identify existence of a particular relationship that links different model constructs. These relations might be recognized between classes (associations, inheritance, etc.), or relationships among operations that represent interaction and message calls. In addition, they may define relationships among different packages, interfaces between the packages, and the content of the packages. Table 8 shows the predicate are used to describe such relationships.

Table 8: Predicate to describe relations between artifacts

Predicate	Meaning
<i>inheritance</i> (C1, C2)	this predicate indicates that Class C1 is a super class of class C2
<i>association</i> (C1, C2)	predicate indicates that class C1 has a reference to another class C2
<i>associationThroughPackages</i> (P1, C1, P2, C2)	this predicate is similar to the one presented above, however here the classes are located in different packages, s.t. $C1 \in P1$ and $C2 \in P2$.
<i>aggregation</i> (C1, C2)	shows that class C1 has a reference to another class C2, where C2 can be seen as a part of C1.
<i>aggregationThroughPackages</i> (P1, C1, P2, C2)	aggregation among two classes , it differs from the previous one in that C1 and C2 are located in different packages P1 and P2 respectively.
<i>realization</i> (C1, I)	this predicate shows that class C1 realizes and implements operations presented in an interface I.
<i>packageOfClasses</i> (C1, [n])	this predicates indicates that one class C1 or more classes, presented as a list [], are grouped together in one package, where ($n \geq 0$).
<i>asscoiationFromClassToFacade</i> (P1, C, P2, F)	this predicate indicates that one class C1, located in package P1 has a reference to an interface F which is located in another package P2.
<i>asscoiationFromFacadeToClass</i> (P, F, P, C)	this predicates indicates that there is a reference from a façade interface F to a class C, where both of them are located in the same package P.
<i>calls</i> (C1, M1, C2, M2)	this predicate shows that a method M1 located in class C1 has a call to another method M2 presented in Class C2.
<i>uses</i> (C1, M, C2, A)	this predicate shows that a method M located in class C1 uses an attribute A that is declared in another class C2.

4.3 Assessing Prolog: Strengths and Weaknesses

Software design patterns reflect best practice solutions applied to frequent design problems. Our initial work focused on developing rules capable of identifying software design patterns. We surveyed Prolog-based approaches used for design patterns in software design [147]. The paper highlighted the capability of Prolog rules in representing and detecting the design patterns. We surveyed and analyzed representative Prolog-based approaches. Furthermore, we conducted an experiment where the structural and behavioral aspects of a number of design patterns were represented using Prolog predicates. Our analysis and evaluation revealed a list of strengths and weaknesses of the Prolog-based detection approaches. Evaluated approaches included some presented in the literature (Section 3.4) as well as our own experiments in this regard.

4.3.1 Strengths

Representational adequacy is the ability of the language to depict all sorts of knowledge required in the described domain. As explained in the previous section, design patterns are represented as rules and the software design is represented as facts. The rules and facts are stored in a Prolog database. When Prolog is used as a repository of design knowledge, a number of features can be gained.

- Prolog rules represent the characteristics and constraints of each design element. Thus, these rules can be utilized to verify the design elements consistencies.
- Prolog can perform exhaustive search for solutions during its execution. As a result, Prolog engine has the ability to detect all occurrences of a specific pattern.

- Logic based languages in general are able to represent the real world more accurately.

Inferential adequacy refers to the capability to drive new knowledge from the available one by manipulating the representational structures.

- Prolog assert and retract clauses can be utilized to add and remove structural facts about design elements.
- Prolog is able to derive new rules from the existing rules contained within the knowledge base.

4.3.2 Weaknesses

Despite the strengths of Prolog rules and the scientific contributions of the Prolog-based studies [108-111], in the following we list several drawbacks of Prolog when detecting design patterns. The weaknesses are discussed in terms of inferential efficiency and acquisitional efficiency. The former refers to the ability to integrate more information into the knowledge structure to focus the attention on a promising way; while the latter indicates the capability of utilizing automatic methods to gain new knowledge when possible instead of human intervention.

- Exact matching has been considered in all the surveyed papers, also in the experiment we conducted. In case of the binary detection, even if high percentage of the design corresponds to the design pattern, it cannot be detected. Moreover, ranking of the detected patterns is not supported. In reality, the fuzzy conditions need to be represented when describing the patterns; also the detection approach should be able to detect the partial matching.

- Another issue related to the design pattern variants. Due to the variations of design pattern instances, it is not easy to decide entirely whether a set of classes in software design is accurately a design pattern instance. The patterns can be formalized by different variants, i.e. each variant is represented by a separate rule even if the variation is minimal. As a result the varying definitions impact negatively the accuracy of pattern detection approaches.
- The results we obtained demonstrated that Prolog engine can detect only the classes represented and their relations using the logical rules. In other words, it is difficult to detect directly similar (not identical) design pattern instances. Thus, we may need to extend the rules to include all the missing design elements. Accordingly, the rules grow rapidly and become impractical to manage.
- The recall and precision measures show good results either in our experiment or in the surveyed studies. However, when referring to the three aforementioned points, these accuracy measures are satisfied with exact matching to the instances occurrences and with the specified variant of design pattern.
- Some design patterns that are similar in some features might be misrecognized as another design pattern. For example, when querying about the occurrence of the adapter pattern, the proxy pattern may be retrieved as adapter. The association between the subclasses is not considered because this relation was not described in the rule. This is because Prolog is based on a closed world assumption. The closed world assumption requires that all the atoms in a domain must be specified. That is, rules that have not been specified will be considered to be there unless we put restriction which is difficult to be specified.

- Prolog-based approaches have no ability to work directly on the given design models. These approaches require pre-processing in order to convert the original model representation into another more suitable representation for the proposed formalism. In this case, there is a need for converting the design patterns into predicates to allow operating on them. Such conversion suffers many drawbacks.
- The logic approaches require skills in mathematics and logic that might not be available or hard to learn by personnel in charge of specifying patterns, making them difficult to use. Pattern designers might not be familiar with the mathematical formalisms, making them difficult to use to specify patterns.
- Most of the surveyed approaches reproduce the design from the source code, which usually doesn't represent the actual design of the system. Not all the relations and interactions are reflected in this conversion. Thus, recovery of design patterns instances and their variants from reverse engineering source code of legacy applications remained challenging due to nonexistence of formal definitions for all patterns and their variants. Such variations hamper the accuracy of pattern detection techniques and tools.

CHAPTER 5

THE RESEARCH PROBLEM AND PROPOSED

SOLUTION

This chapter discusses the research questions to be answered through this dissertation. Then it describes the proposed solution represented by an ILP-based transformation system. The proposed system consists of several components for inducing, applying, evaluating, and validating the transformation rules. In this chapter, Section 5.1 presents the research questions, while Section 5.2 introduces the solution proposed to answer the research questions. The remaining sections of this chapter describe in details the various phases of the proposed solution.

5.1 Research Questions

The objective of this dissertation is to utilize existing analysis-design pairs to generate model transformation rules that can be utilized in future developments. Our literature survey reveals that no effort has been made to generate the rules for the analysis-design transformation. In order to achieve this objective, this thesis, specifically, addresses the following research questions:

***RQ1-** How mapping rules can be derived from examples of traceable analysis-design pairs? How mapping rules can be executed to develop a new design for a given analysis model?*

***RQ2-** What representation should be used for such rules? What corresponding representation can be used to represent the analysis and design models?*

***RQ3-** How can the expert evaluate and update the rules stored in the rule-base?*

Given a collection of pairs of source-target models, the three research questions above are explained in the following.

The first research question (RQ1) is concerned with generating rules that specify mappings between artifacts of source models and their corresponding artifacts of target models in a formal way. The traceability details of available pairs of source-target models help discovering the mapping between the source and target artifacts. The derived rules specify heuristics and patterns to allow transforming a given source model into its corresponding target model. The process of deriving and applying the transformation rules is considered as highly iterative and interactive process. In turn this would guide us to the second research question. RQ2 is about the representation of the extracted rules as well as the representation of the pairs of source-target models. The representation should be efficient and adequate as explained in Chapter 4.

The extracted rules need to be evaluated and updated based on the feedback from experts and/or after applying them, in this regard RQ3 is about evaluating and updating the applied rules. Since the transformation rules cannot be derived completely and perfectly

from the initial examples, it is assumed that experts will evaluate these rules based on some criteria. This may result in adding more rules or assigning weights to the existing rules.

5.2 Solution Approach

The proposed transformation system can be divided into three main components: i) the transformation rules generation and generalization, ii) the transformation rules application and evaluation, and iii) the transformation rules validation and refinement. These three components and other supporting functions are demonstrated in Figure 12.

ILP systems often start with a preliminary pre-processing stage and ends with a post-processing stage. In addition there is an intermediate step between the rules generation and rule application, to translate the generalized rules produced by the used ILP engine into a fit format of fact-based rule language. On the other hand, the post-processing stage concentrates on improving the efficiency by removing the redundant clauses in the induced theory. Hereafter a "theory" is used to refer to a set of induced hypotheses.

It is worth to mentioning that, before starting the process of leaning, the given data is divided randomly into learning and validation sets in a ratio 2:1. In the following we describe all the components and functions employed in the proposed transformation system.

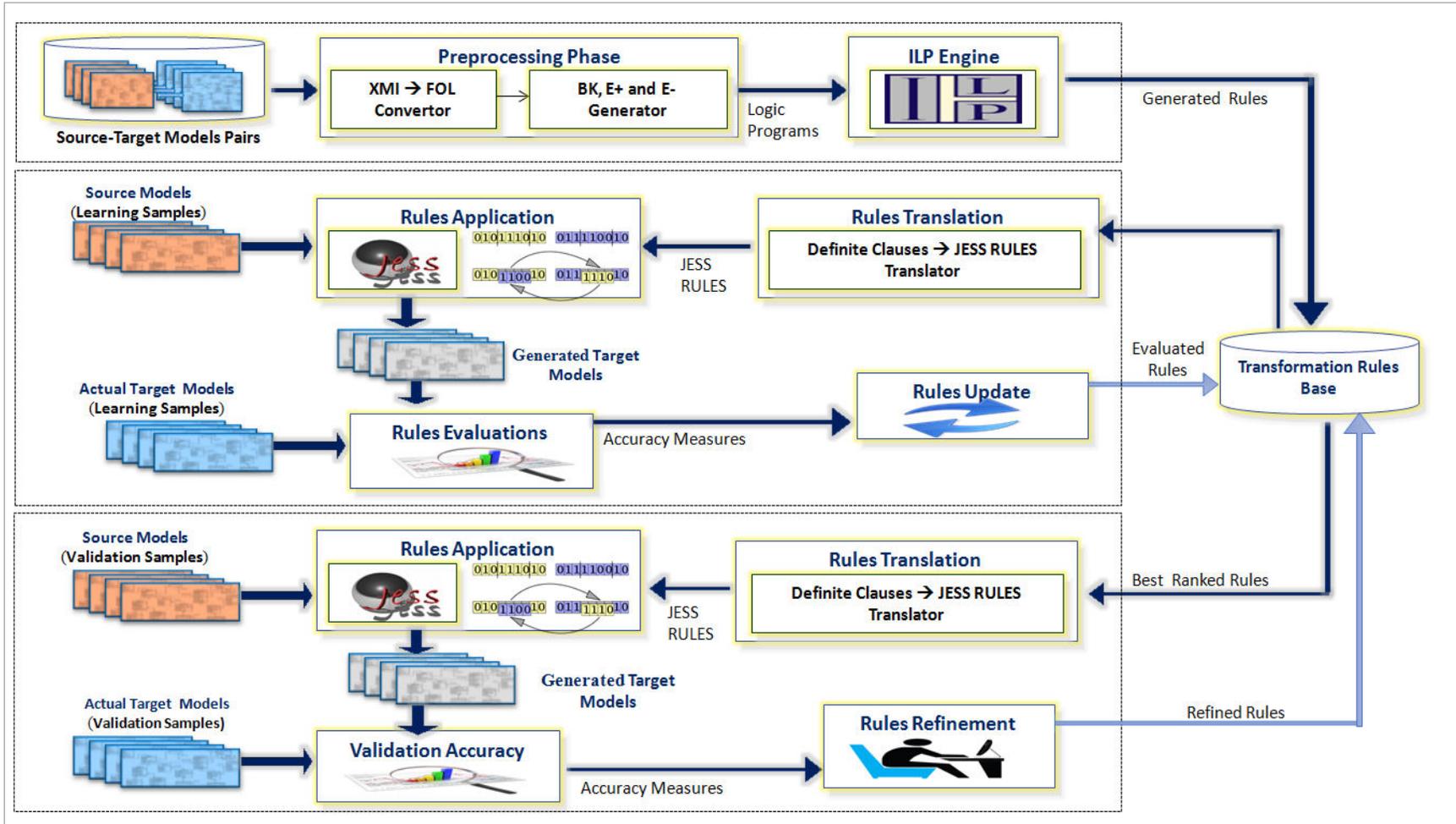


Figure 12: Architecture of the proposed transformation system

5.3 Preprocessing

The given source models (requirement analysis) and target models (software design) are represented in XML-based Metadata Interchange (XMI) format. Thus, we expect that the source and target models are provided as two XMI files. The pre-processing stage focuses on preparing the source and target models to accommodate the input requirements of the employed ILP system. It consists of several procedures detailed in the following.

5.3.1 Background Knowledge and Examples Files Creation

This stage includes also the conversion of the UML models (given in XMI format) into first order predicate logic. In order to achieve this conversion, we implemented a Java tool for this purpose. The conversion tool takes two XMI files; one has the source model artifacts while the second depicts the artifacts presented in the target models. Indeed, many of the artifacts presented in the source models appear again in the target models. Thus, the new target artifacts are considered as the heads of the target relations to be learned.

In this regard, the source model is converted to a separate logic program used as knowledge background, whilst the new artifacts in the target model are converted into logic program that describes the positive examples used for learning. That is, the background facts depict all the source model artifacts, while the new artifacts appear in the target models are represented in the positive examples file. Figure 13 describe how the given files are converted into the required format and what the expected output from the ILP system are.

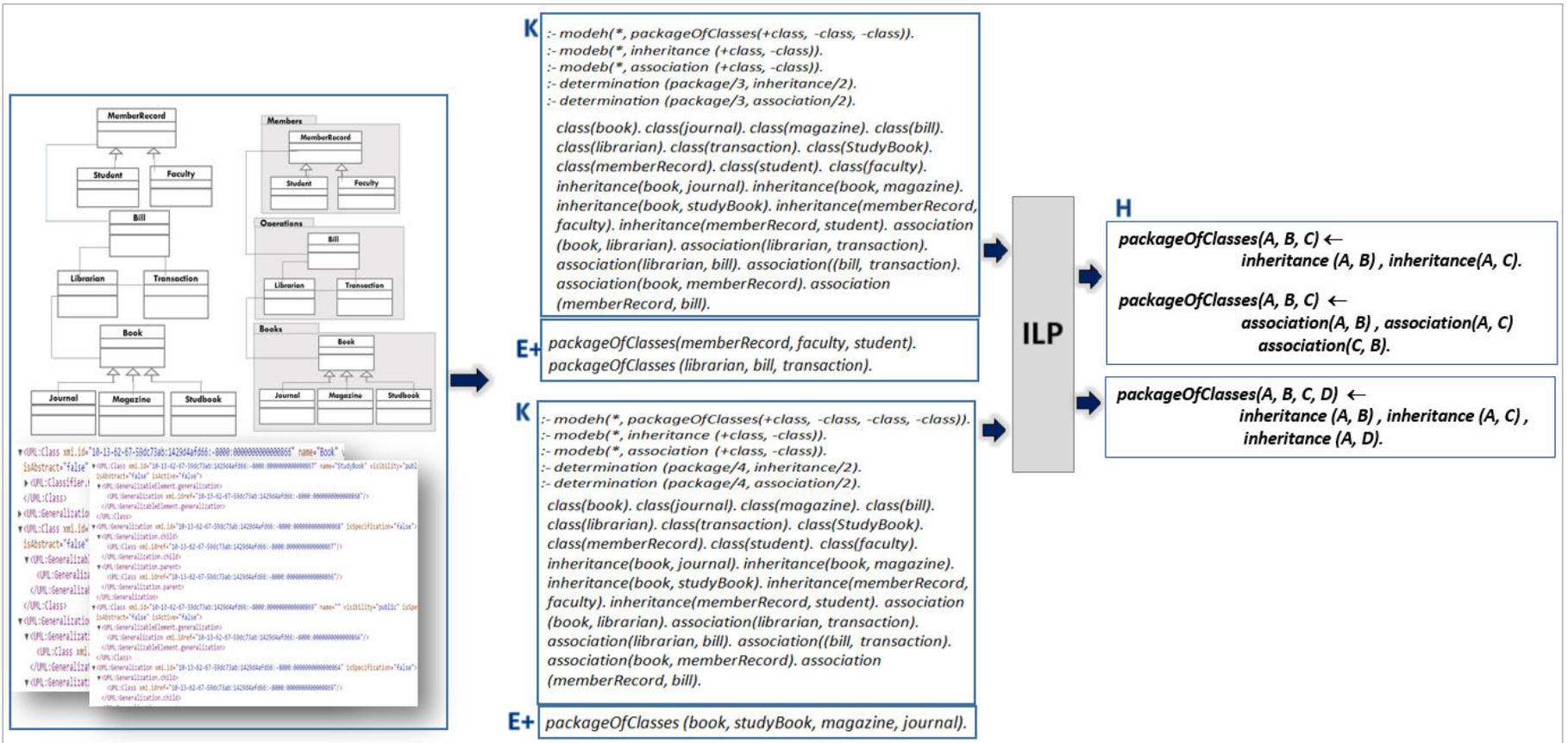


Figure 13: Life cycle of the input files

Essentially, the ILP system requires this information in order to learn the target predicate. For some ILP systems, the negative examples are required to start the learning process. In our context, it is not possible to determine the negative examples from the given models. In such situations, Closed World Assumption [47] can be employed to generate a set of the negative examples based on the given positives.

5.3.2 Grouping of the Positive Examples

It is expected to have different predicates to be considered as positive examples. Each predicate represents a particular artifact found in the target model. The similar atoms in the positive examples file will be grouped together. In other words, the atoms having the same predicate and the same arity (i.e. number and type of arguments) are grouped together. The target of grouping is to present each group individually along with the background knowledge to the ILP system. In each run, one group of the positive examples can be used to induce the target hypotheses. Similarly, when generating the negative examples, the same criteria should be considered.

In ILP systems generally, the positive example represents the head of the target hypothesis. Since we have different predicate there is a need to classify them where each group represent only one positive example type regardless of the number. The steps of this operation are described in the Algorithm 1.

After grouping the positive examples, each group is batched together along with the background knowledge to ILP system to induce hypotheses (rules). For each positives group, the induced rules have the same predicate and arity as the predicate and arity presented in this group of examples.

Algorithm 1: Background knowledge and Examples Creation

Input: Pairs of source & target models $ST = ((s_1, t_1), \dots, (s_n, t_n))$.

Output: Background knowledge K ,
Groups of positive examples PE

```
1:  $B \leftarrow \emptyset$ 
2:  $E \leftarrow \emptyset$ 
// Convert each  $st \in ST$  into logic predicate
3: Repeat Until  $ST = \emptyset$ 
4:   convert  $st_i = \{s_i, t_i\}$  into logic fact such that:
5:      $B \leftarrow B \cup s_i \quad \forall s \in S$ 
6:      $E \leftarrow E \cup t_i \quad \forall t \in T$ 
7:    $ST = ST \setminus (s_i, t_i)$ 
8: end repeat

//Classify the examples in  $E$  into different groups
9:  $PE \leftarrow \emptyset$ 
10:  $i = 1$ .
11: Repeat until  $E = \emptyset$ 
12:   Create new group  $pe_i$ 
13:   Pick the first example  $e_1$  From  $E$  s.t.
      $pe_i = pe_i \cup e_1$ 
      $E = E \setminus e_1$ 
14:   For all the remaining examples in  $E$  ( $e_2 \dots e_n$ )
15:     check similarity of predicate and arity of  $(e_j, e_1)$ 
16:     if similar then
        $pe_i = pe_i \cup e_j$ 
        $E = E \setminus e_j$ 
17:   end for
18:  $PE = PE \cup pe_i$ 
19:  $i = i + 1$ 
20: return  $B$  and  $PE = \{pe_1, \dots, pe_n\}$  ( $1 \leq n \leq E$ )
```

It is important to note that it is not possible to determine the existence of negative examples in the given source-target pairs, if they are not declared explicitly. Thus, the current phase doesn't consider the negatives examples (unless it is generates using CWA). The ILP system is used to learn the relationships among the different artifacts. Then after applying the rules, some negatives cases may appear. Afterwards, the learning process can be repeated again using the positives and negatives along with the background knowledge.

5.4 Rules Induction and Generalization

After converting the given examples (XMI format) into logic programs and grouping the positive examples, the process of generating the rules starts. As explained before, in case of having multiple predicates to be learned each group having the same predicate and arity can be used at a time.

Generally, the transformation problem needs a set of transformation rules in order to cover all aspects of the transformation. The more examples that can be used the more rules can be generalized. In turn more transformation aspects are covered.

In this phase, the target is producing the maximum number of transformation rules in order to cover all the possible mappings between the source model and target model. In this context, the transformation system can be encoded as a set of transformation rules $S=[r_1, r_2, \dots, r_n]$. Each rule can also be encoded as a pair of promise and conclusion $r_i=[P, C]$ where P is the artifacts to be searched in the source model and C is the artifacts that would be instantiated for generating the target model as output. Roughly speaking, the transformation rule here is used to analyze a particular aspect of the source model given as input and synthesize the corresponding target model to be presented as output.

When having more than one group of positive examples, one group along with the background knowledge are presented at a time for learning. That is, the same knowledge background will be used in the different runs. However, most ILP system require the usage of types and mode declarations. That is, the background knowledge needs to be adjusted in each run to accommodate the required modes declarations and determinations. These types of modes are used to specify the structure of the target hypothesis.

Algorithm 2: Transformation Rules Generation

RD= {R,D} Requirements analysis- Software design Pairs
TR: List of derived transformation rules induced from E
B: Knowledge Background
E ={E+, E-} positive and negative examples

Input: Initial set of RD

Output: A set of TR such that TR derived from E and B

```
1.   E+ ← ∅
2.   B ← ∅
// Convert the constructs r and d ∈ RD into a logic predicate
3.   While RD ≠ ∅
4.     convert rdi = { ri , di } into logic predicate such that:
5.     B ← B ∪ ri  ∀ r ∈ R
6.     E+ ← E+ ∪ di  ∀ d ∈ D
7.   End while

// Generate The transformation rules TR
8.   TR ← ∅
9.   Repeat
10.  Create three files required by ALEPH such that:
11.  file.b ⊂ B, file.f ⊂ E+, and file.n ⊂ E-
12.  ALEPH induces a rule R = (LHS, RHS)
13.  TR ← TR ∪ R
14.  Until ∃ R ∈ TR , ∀ di ∈ E+
15.  Return TR.
```

SPL- based systems can be used to induce the expected hypotheses one by one at each run. This approach can give accurate results when the predicates are independent. That is, learning completely the definition of a particular predicate before learning the next one.

When learning the predicates that are dependent means that the ILP system should put into account the order of learning. For instance, the packaging of the classes rule should be ahead of the rule of establishing the interfaces among packages. Algorithm 2 explains the steps used in this stage.

As a result of using ILP engine, based on the given input requirements it is expected to get a list of hypotheses (rules). Each rule represented in the form of definite clause

(*Conclusion* \leftarrow *Premise*). These rules are considered the new knowledge induced by the ILP learner. This knowledge can be applied when presenting new premises to generate corresponding conclusion.

A list of such transformation rules induced by ALEPH is presented in Chapter 6. In addition, in Chapter 8 we present more rules generated using MTILP system.

5.5 Rules Application and Evaluation

As discussed, all the generated rules, initially, are stored in the rule base in the form of logic program (definite clauses). That is, based on the proposed ILP-based transformation system, regardless, of the ILP engine is employed to induce the rules, there is a need to produce executable rules.

5.5.1 JESS Rules Translation

To be able to execute the rules stored in the rule base, a rule-based system JESS [63, 64] is employed for that purpose. It is worth mentioning that, JESS is only the tool used from outside in the ILP-based model transformation system. To be able to apply the rules using JESS engine, the stored rules are translated into JESS rules format.

The JESS program usually is composed of facts and rules. Facts represent the data the rules work with. All the rules define in JESS are stored internally in the JESS rule-base. The facts and rules managed by the rule engine are organized into modules that can be executed at different stages. The rules have two parts: conditions and actions. The former represented by and called left hand side (LHS), while the latter represented by and called

right hand side (RHS). When the LHS, matching fact patterns, is (are) satisfied, the RHS, a list of actions, are performed.

$$IF(fact_1 \wedge fact_2 \wedge \dots \wedge fact_n) THEN (action_1 \wedge action_2 \wedge \dots \wedge action_m)$$

JESS is written in Java, this it is considered as an ideal means for implementing the induced rules to Java-based software systems. Example of the translated rules in JESS format is presented in Figure 20.

5.5.2 Measuring Performance of the Induced Rules

Different measures can be used to evaluate the produced rules and their applicability. Some of these measures can be used before applying the rules and some others can be used after the application of the rules. After learning and before applying the rules, two measures can be used to examine the quality of the induced rule. These measures include completeness and consistency. Completeness indicates that the induced hypotheses along with background knowledge cover all given positive examples. Consistency means that the induced hypotheses along with background knowledge cover none of the negative examples.

For the problem solved by ILP-based systems, usually the performance can be measured by grouping the results as true positive (TP), true negative (TN), false positive (FP) and false negative (FN). In the context of this work, TP refer to the correct target model artifacts generated by the transformation rules. FP refer to the incorrect target model artifacts generated by the transformation rules. FN refer to the correct target model artifacts in the actual target model, but the transformation rules do not generate. TN refer to the other model target artifact that were correctly ignored by the transformation rules

(in grouping problems). Table 9 demonstrates the measures collected, through this work, to measure the induced rules performance.

Table 9: Performance Measurements

Metric	One artifact	Collection of artifacts
Accuracy	Accuracy $p_j = 100 \frac{(TP_{p_j} + TN_{p_j})}{(TP_{p_j} + TN_{p_j} + FP_{p_j} + FN_{p_j})}$	<i>Accuracy</i> = $\frac{\sum_{m=1}^{i=1} Accuracy_{p_i}}{n}$
Specificity	Specificity $p_j = \frac{TN_{p_j}}{TN_{p_j} + FP_{p_j}}$	<i>Specificity</i> = $\frac{\sum_{m=1}^{i=1} Specificity_{p_i}}{n}$
Precision	Precision $p_j = \frac{TP_{p_j}}{TP_{p_j} + FP_{p_j}}$	<i>Precision</i> = $\frac{\sum_{m=1}^{i=1} Precision_{p_i}}{n}$
Recall	Recall or Sensitivity $p_j = \frac{TP_{p_j}}{TP_{p_j} + FN_{p_j}}$	<i>Recall</i> = $\frac{\sum_{m=1}^{i=1} Recall_{p_i}}{n}$
F-measure	F – measure $p_j = \frac{2 * Precision_{p_j} * Recall_{p_j}}{Precision_{p_j} + Recall_{p_j}}$	<i>F – measure</i> = $\frac{\sum_{m=1}^{i=1} F - measure_{p_i}}{n}$

We validate the generated artifacts by comparing them with the actual artifacts. Different measures, shown above, have been collected in our experiments. For instance, when validating the generated packages, we need to pay attention to their content. Let $AD = \{p_1, p_2, \dots, p_n\}$ be the number of packages of the actual design. Let $GD = \{p_1, p_2, \dots, p_m\}$ be the number of packages of the corresponding rule-created design, where m could be less than, equal to, or greater than n. In AD each p_i consists of

a number of classes $c_{i1}, c_{i2}, \dots, c_{ik}$ and the corresponding package p_j in GD may consist of the same, more or less classes $c_{j1}, c_{j2}, \dots, c_{jl}$. TP refers to the correctly placed classes in the created package p_j while TN refers to the classes that are not placed in p_j correctly. FP indicates the extra classes placed in p_j while they are not presented in p_i . Finally, FN indicates the number of classes that exist in p_i but not placed in p_j . Finally, we calculate the average across all packages for all the measures.

To calculate the overall recall and precision measures, we calculate the two measures for each package then calculate the average across all packages as shown in the third column in Table 9.

For the Façades experiments it was not applicable to calculate TN so we exclude some measures. Thus, only precision, recall and f-measure have been considered. We calculate the precision by dividing the number of the correct rule-created interfaces by the overall number of rule-created interfaces. On the other hand, we calculate recall by dividing the number of the correct rule-created interfaces by the total number of interfaces that exist in the actual design.

5.5.3 Application and Evaluation using a GA-based procedure

We built a java application that includes all generated rules in the form that is ready to be triggered. When a new instance (i.e. source model for new system) is presented, the models are converted to logic program (all the source artifacts are represented as facts). A rule will fire if the facts that satisfy its conditions exist. Firing a rule means some facts are asserted or some others may be retracted. After running the application on the given

source model, the obtained facts are supposed to represent the corresponding target model. To evaluate the induced rules we use different procedures explained in the sequel.

5.5.3.1 Procedure 1: Individual Rule Application on All Source Models:

In this procedure, all the source models from different examples are batched as one file. Then the induced rules are applied on the given models one by one. That is, applying the rules one by one, say r_i , on the batched source models. The resulting target models from each run can be used to evaluate the accuracy of the applied rule r_i . The generated target models can be compared to the expected (actual) target models when having the source-target pairs already available for learning. In other words, the capability of each rule is assessed by measuring the generated and the actual target models distance. For this purpose, various accuracy measures are used (next section introduces more details about the used measurements).

The aim of this procedure is to assess to which extent the induced rule is generic and is able to be applied alone to different instances. Consequently, the rules are ranked based on their accuracy and a threshold can be used to eliminate the rules that were not able to show the minimum expected performance.

5.5.3.2 Procedure 2: GA-based Rules Application on an Individual Learning System

The aim of this procedure is to apply a combination of rules to find the best accuracy for each learning system. To implement this, a genetic algorithm was employed to find the best combination. The procedure starts by generating a random (binary) population. The number of chromosomes equals a fixed value while the number of the genes equals the

number of rules to be evaluated. Figure 14 shows an example of a chromosome representation used in this procedure. It is a binary representation, 1 means the corresponding rule is active and ready to be applied while 0 means inactive rule.

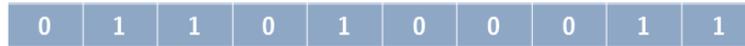


Figure 14: Example of a chromosome representation

For instance, when presenting the transformation rules and the chromosome shown in the figure, the rules having the numbers 2,3,5,9 and 10 will be applied on the given source model. After rules application, for each chromosome, a fitness function is calculated. The accuracy measure is considered as the fitness function here.

All the generated chromosomes are considered for application and the best accuracy are kept. Then crossover and mutation are performed to generate the next generation. This procedure is repeated in order to get the best combinations of the rules.

As a result, for each presented system it is supposed to find the best combination of rules that give us the best accuracy. We utilize this knowledge to specify the frequency of each rule application, i.e. we track how many times each rule has appeared in the best combinations. We find the percentage of rule application as follows:

$$\textit{Percentage of applications times} = \textit{number of applications times} / \textit{number of systems}$$

The frequency is used to rank the rules. The highest frequency refers to the most applied rule to give the best accuracy for the presented systems. This means the top ranked rules have a higher chance to be applied on the systems on the next phase. Algorithm 3 presents the steps of this application and evaluation phase.

Algorithm 3: Transformation Rules Evaluation

RQ: New instance of requirements analysis (XMI format)
SD: Corresponding software design (XMI format)
TR: List of derived transformation (Clauses)
JR: List of the transformation rules in JESS script
F: A set of predicates representing RQ artifacts Facts
INPUT: A new instance of RQ
OUTPUT: SD corresponds to given RQ

```
// Translate TR into JESS rules
1. JR ← ∅
2. While TR not empty do
3.   ∀ tri ∈ TR translate tri into jri
4.   JR ← JR ∪ jri where jri = { P, C }
5.   TR ← TR / tri
6. End while
// Convert the given requirement models into logic predicates
7. F ← ∅
8. While RQ not empty do
9.   ∀ rqi ∈ RQ convert rqi into a predicate fi
10.  F ← F ∪ fi
11.  RQ ← RQ / rqi
12. End while
13. Feed F and JR to the Transformation System (Java application + Jess Engine)
14. Based on F, for each jri whenever P is satisfied => Fire jri
15. New Fi is asserted F ← F ∪ fi   ∀   existing fact fi is removed
16. Convert resultant F to XMI format (SD)
```

5.6 Rules Validation and Refinement

The rules evaluation conducted in the previous phase uses the learning samples, used for rules induction, to evaluate the rules in various ways. As a result, it ends up with a set of best rules ranked based on their frequency of application.

Furthermore, in the validation phase, the transformation designer can validate the induced rules correctness by applying them on more examples as test cases. Thus, the best ranked rules are tested using the predefined set of validation samples (unseen examples). Where the two ranked rules are applied against the validation system, then one more rule is

added to the set of applied rules in each iteration. The generated target models can be compared to the expected (actual) target models when having the source-target pairs already available for validation. In this case, a combination of the ranked best rules are assessed by measuring the generated and the actual target models distance. Another procedure is used in the experiments to consider the application of the best rules randomly selected. The aim of this phase is to investigate the capability of the induced rules to achieve the transformations in a future development.

Different measurements are used to measure the performance of the rules when using unseen examples (see Section 5.5.2). Furthermore, in this phase human expert evaluation for the resulted target models can be considered. This type of evaluation may help to update the priority of the rules application. In addition, expert opinions may contribute to the rule base by adding new rules or relaxing the application of other rules.

Based upon the aforementioned evaluation ideas, the transformation designer is allowed to refine the transformation rules manually by adding some relations and removing others. In reality, the process of reviewing and updating the rules is as difficult as authoring and generating them [105]. In our system, the refinement process has different facets as follows: a) assigning weights to the applied rules where the higher weight the more frequent application, b) proposing new rules or adding conditions or constraints to apply some rules, c) refining the existing rules by adding more relations or extracting others.

5.7 Post-processing

After evaluating and validating the generated transformation rules, the final refined rules are stored in the rules base. However, the obtained rules would be added to a set of transformation rules which already exist in the base. Since, the new rules came from examples, similar examples or transformation aspects might be presented before. That results in similar rules that might be already stored.

Therefore, this phase simply focuses on two tasks. First, eliminating the repeated rules if similar rules already exist in the base. Second, finding if the new rules can be subsumed by other rules already stored in the base.

For instance, when having two rules R1 and R2 where they have the same head atom regardless of the arity. The rule R1 is considered subsumed by rule R2 if the arity of R2 is less and the type of body atoms are similar. Figure 15 demonstrates an examples of two rules for introducing packaging to class diagrams.

<pre>packageOfClasses(A, B, C, D) ← class(D) , class(C) , class(B) , class(A) , inheritance(A, C) , inheritance(A, D) , inheritance(A, B).</pre>	<pre>packageOfClasses(A, B, C, D, E) ← class(E) , class(D) , class(C) , class(B) , class(A) , inheritance(D, A) , inheritance(D, B) , inheritance(D, E) , inheritance(D, C).</pre>
Rule1	Rule 2

Figure 15: Examples of rules to be subsumed

CHAPTER 6

INDUCTION USING ALEPH SYSTEM

The transformation system shown in Chapter 5 proposes using an ILP system as an engine to induce hypotheses from the available transformation knowledge represented by examples. For this purpose, one of the common and widely used ILP systems, called ALEPH, has been selected. It implements several algorithms and ideas in the field of ILP and can actually simulate the behavior of other learning systems such as PROGOL.

This chapter presents the experimental work performed to generate transformation rules. ALEPH system is employed, as the ILP engine, in the proposed transformation system described in Chapter 5. The objectives of the experiments conducted using ALEPH system are:

Obj1: How good is the performance of ALEPH system in transformation rules induction?

Obj2: Is ALEPH enough to induce MDD transformation rules??

6.1 ALEPH Overview

ALEPH has been widely used in solving various types of problems in the literature. It has implemented different evaluation functions and search strategies. The purpose of ALEPH

is to generalize the given examples and the background knowledge into a hypothesis that covers most of the positives and avoids the negatives.

The strategy of most ILP systems is to generate a hypothesis based on a particular predicate. The term “hypothesis” is used to refer to a single clause. To induce another predicate there is a need to prepare the input files at each run to accommodate the intended predicate.

ALEPH system requires three types of input to induce the target theory. The input files consists of background knowledge, positive examples, and negative examples. Thus it is needed to generate the input files to accommodate the declarations requirements at each action.

6.1.1 ALEPH Input Requirements

By using ALEPH system, an independent run is performed to produce a single rule, i.e. a single hypothesis from the given examples with background knowledge. A successful ALEPH induction requires the preparation of three data files to construct theories.

In each run, there is a need to feed three files containing the knowledge background, the positive and the negative examples to the ALEPH system. The facts representing the given models are included in the background knowledge file (*.b). Based on the given mapping for each new construct appears in the source model (software design) the positive examples are represented in the file (*.f) specialized for the positive examples E^+ . Similarly, the negative examples are included another file (*.n). First, the three files are fed into ALEPH system for automatic extraction of mode and type information from the provided background knowledge.

6.1.1.1 Background Knowledge

Two parts included in the background knowledge: rules structure and artifacts descriptions. The former is used to guide the construction of a single rule. The latter is the representation of requirement analysis (source models) artifacts using the predicates defined for the background knowledge representation. In other words, in addition to Prolog predicates, ALEPH requires that some restrictions are placed on the structure of the rule or rules being induced. Besides the background knowledge and a set of examples (represented as facts), ALEPH requires a set of mode declarations (*modeh* and *modeb*) to be appended in the background knowledge file. The purpose of the modes is to define the structure of the intended hypothesis.

As explained in Chapter 2, most of the ILP systems, including ALEPH [36], require to declare modes to constrain the search space. ALEPH also requires the use of a special predicate called determination which restricts possible valid relationships between target and background predicates. For instance, when using ALEPH, the user is requested to include in the background knowledge file the following:

- **Mode declarations.** ALEPH requires specifying two types of modes: *modeh* to define the head of the intended rule, and *modeb* to determine the predicates involved in the rule body. In addition, it is require to specifying the number of successful calls to every predicate and for every argument of each predicate, if this is an input or an output argument. For example, the declaration:

```
:-modeh(*; packageOfClasses(+class;-class;-class)).
```

```
:-modeb(*; inheritance(+class;-class)).
```

That means the “*packageOfClasses*” predicate can be successfully called many times, and that the first argument is input arguments of type “*class*”, whereas the last two arguments are output.

- **Type Specification.** For every argument presented in the modes declaration there is a need to specify the types of all predicates used in the hypothesis construction. For ALEPH, types are just names, and no type-checking is done. Variables of different types are treated differently.
- **Determination statements.** These statements are related to the hypothesis construction process. The user should know beforehand the predicates used in the body of a clause with a specific head predicate. Apart from that, for every background knowledge file in ALEPH (and for every run) it is allowed to use a determination statement for only one head predicate. So, if the user wants to learn clauses for many different head predicates, he needs to prepare separate files with separate determination statements for each head predicate. As it is obvious, ALEPH is trying to restrict the search space by restricting the candidate relations that can possibly participate in the body of the clause. So, for example if we are interested in learning the target relation *packageOfClasses*, the determination statement might include the relations *inheritance*, *association*, etc. because the user assumes that they might be relevant, or because he is sure in some way that they can indeed form some good clauses like the following:

:- determination(packageOfClasses /3,inheritance/2).

:- determination(packageOfClasses /3, association /2).

6.1.1.2 Positive and Negative Examples

It is expected to have different predicates to be used as positive examples. ALEPH system supports a single predicate learning, i.e. there is a need to declare only one predicate at each run. Thus, the learning process is run iteratively till using all the given examples. Only one group of positive examples, that uses the same predicate and arity, can be used in each run. The proposed transformation system provides a function for grouping the positives to be fed the ILP system according to its procedure.

However, some of these predicates are dependent, i.e. the ILP system should take into account the order of learning. For instance, the packaging of the classes rule should be ahead of the rule of establishing the interfaces among packages.

As discussed in Chapter 5, the negative examples are not usually provided either due to the nature of the problem or the lack of data. ALPEH system provides an option to learn using positive examples only. Based on that and similar situations in the literature, we have two options, either generate negative examples by using CWA or learning by positive examples only [148].

Indeed, in the context of the conducted experiments ALEPH in some cases was not able to induce rules using the positive examples only. Thus, there was a need to support the learning process by a set of generated negative examples. CWA was employed to create negative examples.

However, in many learning cases, using CWA is not practical. The reason behind that, to be able to generate the correct set of negatives, it is required to provide a complete set of the positives [149].

6.2 Representation of Model Transformation Problem

As demonstrated in Chapter 5, it is expected to receive the model inputs in XMI format. Thus, we exploited the pre-processing phase to prepare the given XMI models to satisfy ALEPH input requirements. The following sections provide a brief description of the transformation considered and presents the problem and solution representations.

6.2.1 Sample Transformation Tasks

Early on we discussed the difficulty of automating the transformation from requirements analysis to software design [25]. This chapter introduces experiments to induce transformation rules of two investigated transformation tasks. The first task focuses on packaging classes in a class diagram, while the second considers introducing interfaces (i.e., facades) to packages. In the following we describe each case study briefly.

6.2.1.1 Packaging Class Diagram

One of the common tasks when moving from analysis to design is the task of structuring the system classes into packages [150]. During the analysis phase, the class diagram depicts all the classes used in the system and the relations between them. The aim is to develop highly cohesive and loosely coupled packages.

When considering this transformation aspect, we use our proposed approach to learn packaging rules from given analysis-design pair examples. Together Figure 16 and Figure 17 represent a simple example of the source-target pair. They show the analysis model of one of the used examples along with the corresponding initial design model respectively. The initial design model shows the analysis model after introducing the packages.

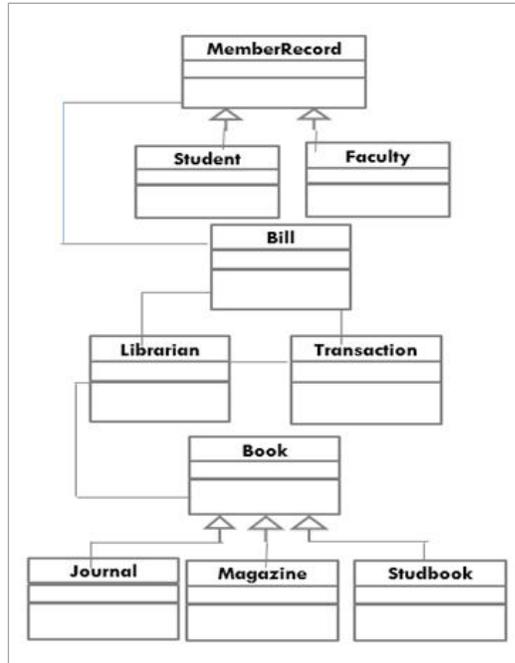


Figure 16: The UML class diagram for analysis-design pair (Source model)

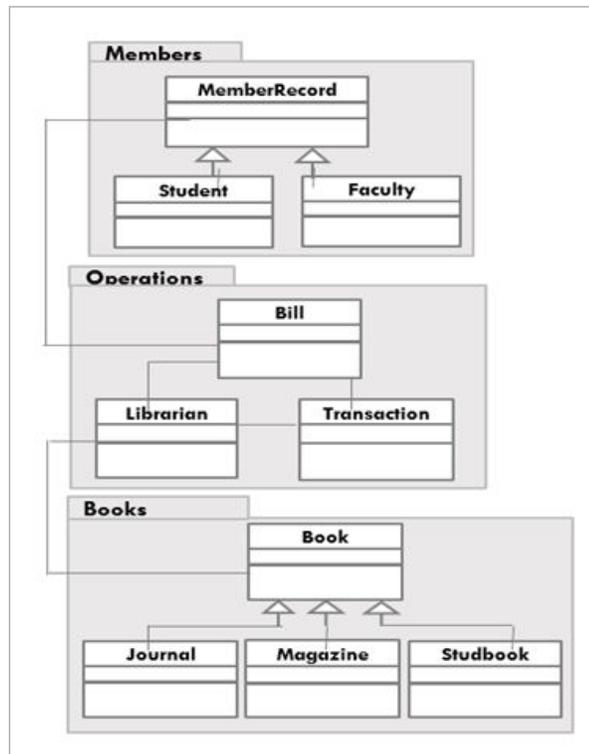


Figure 17: The UML class diagram for analysis-design pair (Target model)

6.2.1.2 Introducing Façades

Introducing façades is another high level software design activity. It is common for a class in one package to have external relations with classes in other packages. The Façade design pattern is used to simplify the interaction process and improve the overall design coupling and cohesion. A façade provides a one “point of contact” to a package of classes (i.e., component). It hides the implementation of the component from its clients, making the component easier to use. In addition, it results in loosely coupled software.

Figure 18 and Figure 19 depict one of such examples. It shows a class diagram that has many inter-package relationships, making the design highly coupled and less maintainable. To overcome this problem the designer introduces façades as another step of transformation from requirement analysis to software design.

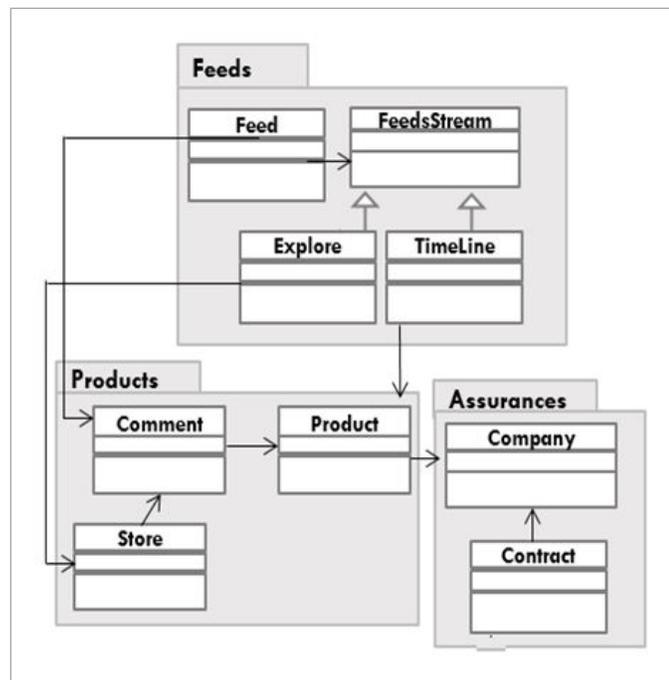


Figure 18: Example of Introducing Facade (Source Model)

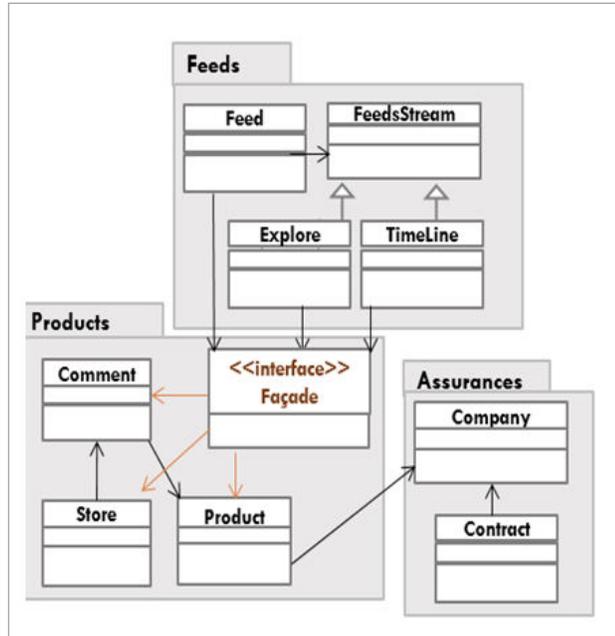


Figure 19: Example of Introducing Facade (Target Model)

6.2.2 Problem and Solution Representations

The given UML models are presented in XMI format. To induce a general hypothesis using ALEPH the problem, represented by XMI models, is converted to logic programs that describe the background knowledge and the positive examples.

Table 10 demonstrates the representation of the packaging problem represented by UML models shown in Figure 16 and Figure 17. The problem representation includes the background knowledge, positive and negative examples. The background knowledge facts represent all artifacts presented in the source model. On the other hand, the positive examples atoms represent only the new artifacts that appear in the target model. The negative examples were introduced as a false combination of the classes used as arguments of the positive examples. Finally, the modes and determinations statements are relevant to the expected hypothesis to be learned.

Table 10: Packaging problem representation written in ALEPH (i.e., Prolog Syntax)

Input Type	Logic Program statements
Types and Modes Declarations	<pre>:- modeh(*, packageOfClasses(+class, -class, -class)). :- modeb(*, inheritance (+class, -class)). :- modeb(*, association (+class, -class)). :- determination (package/3, inheritance/2). :- determination (package/3, association/2). :- modeh(*, packageOfClasses(+class, -class, -class, -class)). :- modeb(*, inheritance (+class, -class)). :- modeb(*, association (+class, -class)). :- determination (package/4, inheritance/2). :- determination (package/4, association/2).</pre>
Background Knowledge	<pre>class(book). class(journal). class(magazine). class(bill). class(librarian). class(transaction). class(StudyBook). class(memberRecord). class(student). class(faculty). inheritance(book, journal). inheritance(book, magazine). inheritance(book, studyBook). inheritance(memberRecord, faculty). inheritance(memberRecord, student). association (book, librarian). association(librarian, transaction). association(librarian, bill). association((bill, transaction). association(book, memberRecord). association (memberRecord, bill).</pre>
Positive Examples	<pre>packageOfClasses(memberRecord, faculty, student). packageOfClasses (librarian, bill, transaction). packageOfClasses (book, studyBook, magazine, journal).</pre>
Negative Examples	<pre>packageOfClasses (memberRecord, studyBook, bill). packageOfClasses(book, student, librarian). packageOfClasses (magazine, faculty, transaction).</pre>

The positive examples consist of two groups, i.e. only one group can be used for leaning by ALEPH system at iteration. This is due to the need to change the declarations, e.g. different declarations are explained in the first row of Table 10.

The representation of the second transformation problem (introducing Façade) is demonstrated in Table 11. It shows a conversion of the UML source-target models presented in Figure 18 and Figure 19.

Table 11: Introducing Facade - Problem representation written in ALEPH (i.e., Prolog Syntax)

Input Type	Logic Program statements
Types and Modes Declarations	<pre>:- modeh(*, packaheHasFacade(+package, interface)). :- modeb(*, packageHasClass(+package, +class)). :- modeb(*, packageHasClass(-package, -class)). :- modeb(*, associationAcrossPackages(-package, -class,+package, +class)). :- determination (packaheHasFacade/2, associationAcrossPackages/4). :- determination (packaheHasFacade/2, packageHasClass/2).</pre>
Background Knowledge	<pre>package(feeds). class(feed). class(feedsStream). class(explore). class(timeLine). association(feed, feedsStream). inheritance (feedsStream,timeline). inheritance(feedsStream, explore). package(prodcuts). class(comment). class(store). class(product). association(comment, product). association(store, comment). package(assurance). class(company). class(contract). association (contract, company). associationAcrossPackages (feeds, feed, products, comment). associationAcrossPackages (feeds, explore, products, store).</pre>
Positive Examples	<pre>packaheHasFacade(products, façade).</pre>
Negative Examples	<pre>packaheHasFacade(feeds, façade). packaheHasFacade(assurance, façade).</pre>

As explained above, it is required to declare the modes and determinations besides the background knowledge and the given examples. Only one package has a façade interface in the given examples, thus the other packages are considered as negative examples when having facades interfaces.

The first column, in Table 12, represents samples of the rules produced by ALEPH system based on the shown examples (and similar ones). In the first three rules, LHS represents the conclusion (e.g. introduce a package to a set of classes) in order to group different classes into a single package wherever RHS which is the premise is satisfied. While the last rule states that a facade is added to a package whenever there is an external reference to a class located in that package.

Table 12: The Solution Representation

Induced Rule	Meaning
$packageOfClasses(A,B,C) \leftarrow inheritance(A,B), inheritance(A,C).$	When there is an inheritance relation between the classes A and B, these two classes are grouped together in one package.
$packageOfClasses(A,B,C) \leftarrow association(A,B), association(A,C), association(B,C).$	When three classes A, B and C have association relations, such that class A is linked to class B, and Class C, also Class B is linked to Class C, and then the three classes can be grouped in one package.
$packageOfClasses(A,B,C,D) \leftarrow inheritance(A,B), inheritance(A,C), inheritance(A,D).$	When three classes A, B, C and D have the presented relationships, such that class A is the parent of classes B, C and D then the four classes can be grouped in one package.
$packageHasFacade(A,B) \leftarrow packaheHasClass(C,D), associationAcrossPackages(C,D,A,E).$	When there is a class D, which is placed in a package C, has a reference to another class E placed in a different package A, a façade interface B is introduced to the destination package A.

To be able to execute the induced rules on a new instance of source analysis models, the JESS rule engine is used to implement them. In other words, the rules represented in JESS language are applied on the given source model to generate the target model. Figure 20 shows some of the induced packaging rules translated in JESS format.

```
Jess> (defrule introduce-Package-Association
?class1 <- (class)
?class2 <- (class)
?class3 <- (class)
  (and (association{(#firstClass == class1.#name &&
                    #secondClass == class2.#name)})
        (association{(#firstClass == class1.#name &&
                    #secondClass == class3.#name)})
        (association{(#firstClass == class2.#name &&
                    #secondClass == class3.#name)}))
=> (assert (packageOfClasses(#first ?class1.#name) (#second ?
class2.#name) (#third ?class3.#name))
))

Jess> (defrule introduce-Package-Inheritance
?class1 <- (class)
?class2 <- (class)
?class3 <- (class)
  (and (inheritance{(#parent == class2.#name && #child ==
                    class1.#name)})
        (inheritance{(#parent == class3.#name && #child ==
                    class1.#name)}))
=> (assert (packageOfClasses(#first ?class1.#name) (#second ?
class2.#name) (#third ?class3.#name))
))
```

Figure 20: Sample rule implemented by JESS rule engine

When applying the implemented rules on new source model, which contains a flat class diagram, we expect that some package artifacts will be added as new predicates to the working memory. These packages group the presented classes as a result of this transformation step. For instance, when applying the rules on a source model, the resultant transformation will include all the given facts as well as the suggested predicates for grouping the classes into packages. This step of transformation doesn't specify a name for the proposed package the designer can put the appropriate name.

6.3 Experiments Setup

The experiments conducted in the following focus on using the ILP-based transformation system to induce transformation rules for the two described tasks (i.e. packaging classes in a class diagram and introducing facades to packages). ALEPH system is the employed ILP engine to learn the rules. Each of the two conducted experiments has two phases: learning and validation. In the learning, a number of examples have been used to generate the initial set of rules. Then the validation samples are used to validate the rules. In the following we describe the datasets used for learning and validation.

6.3.1 Datasets

The datasets used in the experiments comprise 34 systems, each consisting of the analysis and design models. These systems were collected from different sources including examples from text books, academic projects, and by reserve engineering. Each system consists of analysis/design pair. In turn, each design system comprised at least 3 packages. The total number of packages in the base is 218 while the total number of classes and interfaces is 1085. Table 13 shows brief statistics of the systems' artifacts i.e., packages, classes, interfaces, and relationships (including association, aggregation, generalization, and realization).

Table 13: The datasets Statistics

Software artifacts	Min	Max	Mean	Total
Packages	3	27	6	218
Classes and Interfaces	10	151	32	1085
Relationships	11	188	45	1543

Table 14 shows more information about the number of artifacts in each system. The presented statistics are taken from the target models to show a complete picture about the packages and interfaces included in the used systems.

Table 14: Description of the used datasets

Code	Packages	Classes - Interfaces	Relationships	Code	Packages	Classes - Interfaces	Relationships
<i>Sys01</i>	6	22	40	<i>Sys18</i>	12	77	95
<i>Sys02</i>	3	10	17	<i>Sys19</i>	4	18	46
<i>Sys03</i>	7	33	43	<i>Sys20</i>	7	34	41
<i>Sys04</i>	3	12	20	<i>Sys21</i>	4	18	46
<i>Sys05</i>	3	12	14	<i>Sys22</i>	17	91	105
<i>Sys06</i>	27	151	188	<i>Sys23</i>	6	35	60
<i>Sys07</i>	5	25	38	<i>Sys24</i>	3	10	16
<i>Sys08</i>	3	10	12	<i>Sys25</i>	3	13	23
<i>Sys09</i>	5	20	19	<i>Sys26</i>	3	15	27
<i>Sys10</i>	3	11	16	<i>Sys27</i>	6	33	45
<i>Sys11</i>	5	25	29	<i>Sys28</i>	22	110	143
<i>Sys12</i>	4	23	45	<i>Sys29</i>	7	36	50
<i>Sys13</i>	3	11	16	<i>Sys30</i>	6	35	57
<i>Sys14</i>	4	20	32	<i>Sys31</i>	5	26	50
<i>Sys15</i>	5	20	28	<i>Sys32</i>	3	10	11
<i>Sys16</i>	7	41	62	<i>Sys33</i>	6	24	35
<i>Sys17</i>	6	31	49	<i>Sys34</i>	5	23	25

The overall goal of using ILP is to induce the best hypothesis which is able to classify future unseen examples. Thus, the hypotheses generated in the learning phase need to be evaluated in the validation phase. The most common evaluation method used is to evaluate the hypothesis against an independent validation set (a set of labeled examples which is not used in the learning phase). In our conducted experiments, the data is divided 2:1 ratio (learning: validation) in each experiment. Both the learning and validation examples are randomly sampled in such a way that all examples are properly represented.

6.3.2 ALEPH Settings

There are many settings can be adjusted when using ALEPH. To conduct the experiments explained in this section, we started the experiments using the default settings. Then different settings have been adjusted such as search strategies; however the results obtained were comparable.

6.4 Experimental Results / Quantitative Validation

This section shows the results obtained from the two conducted experiments by using the described datasets. In all experiments each set of examples was divided into learning set and validation set in the ratio 2:1. Each set has been selected randomly while ensuring that no system has been selected twice. During learning phase, all the 22 learning systems have been used as input to generalize a set of transformation rules that have been evaluated later in two ways to select the best rules. Then the final rules were validated against the validation set which consists of 12 systems. Samples of the induced transformation rules for the packaging problem are demonstrated in Table 15.

Table 15: Samples of the induced transformation rules using ALEPH- Packaging Problem

Induced Transformation Rule
$packageOfClasses(A,B) \leftarrow association(A,B).$ $packageOfClasses(A,B) \leftarrow inheritance(B,A).$
$packageOfClasses(A,B,C) \leftarrow association(A,B), association(C,A).$ $packageOfClasses(A,B,C) \leftarrow inheritance(B,A), association(A,C).$ $packageOfClasses(A,B,C) \leftarrow inheritance(C,A), inheritance(C,B).$ $packageOfClasses(A,B,C) \leftarrow association(B,A), association(C,A).$
$packageOfClasses(A,B,C,D) \leftarrow association(A,C), association(C,B), association(A,D).$ $packageOfClasses(A,B,C,D) \leftarrow association(A,D), association(C,A), association(C,B).$ $packageOfClasses(A,B,C,D) \leftarrow inheritance(B,A), inheritance(B,C), inheritance(B,D).$ $packageOfClasses(A,B,C,D) \leftarrow association(C,A), association(C,D), association(B,C).$

In addition, Table 16 shows the unique transformation rule induced by ALEPH system for the task of introducing façade. This issue will be discussed further in the next sections.

Table 16: Samples of the induced transformation rules using ALEPH- Introducing Facade

Induced Transformation Rule
$packageHasFacade(A,B) \leftarrow packageHasClass(C,D), associationAcrossPackages(C,D,A,E).$

6.4.1 Performance of the Packaging Rules

We measured the performance of the induced rules individually. The performance of each rule has been measured by applying the rule on all systems in one run. As demonstrated in Section 5.5, different evaluation procedures have been used to measure the rules performance.

Figure 21 shows the performance of each rule when applying them individually on all the learning system, batched together. Some rules, e.g. rules 9-10, show low performance, thus these rules have been retracted from the rules base to avoid impacting the overall system performance.

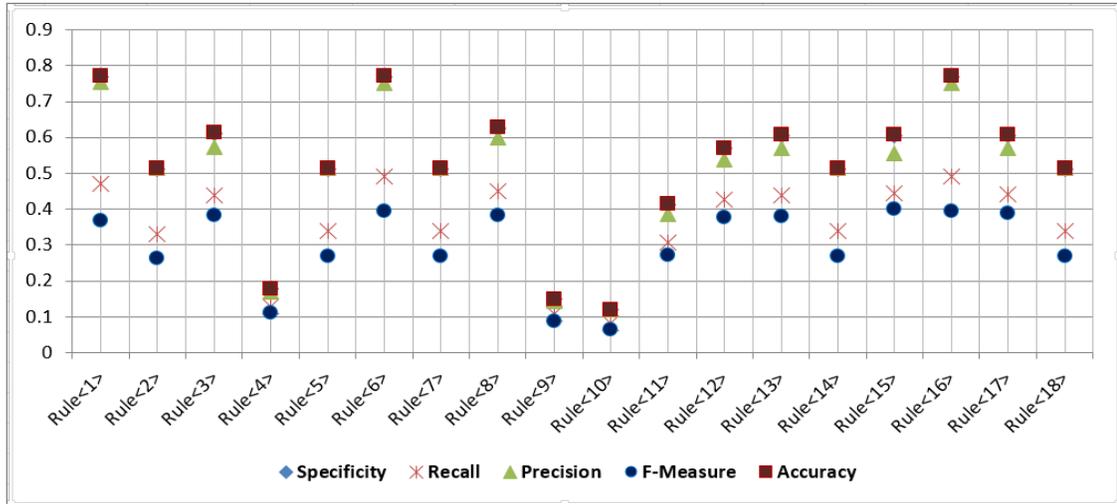


Figure 21: Performance Measures - Rules application on batched learning systems

Then the remaining rules are measured in another way around where a genetic algorithm-based procedure is used to find the subset of rules can give the best results when presenting the learning systems one by one. Figure 22 presents the accuracy measures for each learning system. These experiments were considered only for the packaging case study. In this experiment we paid attention for the number of times each rule has been considered to give the best accuracy with each system.

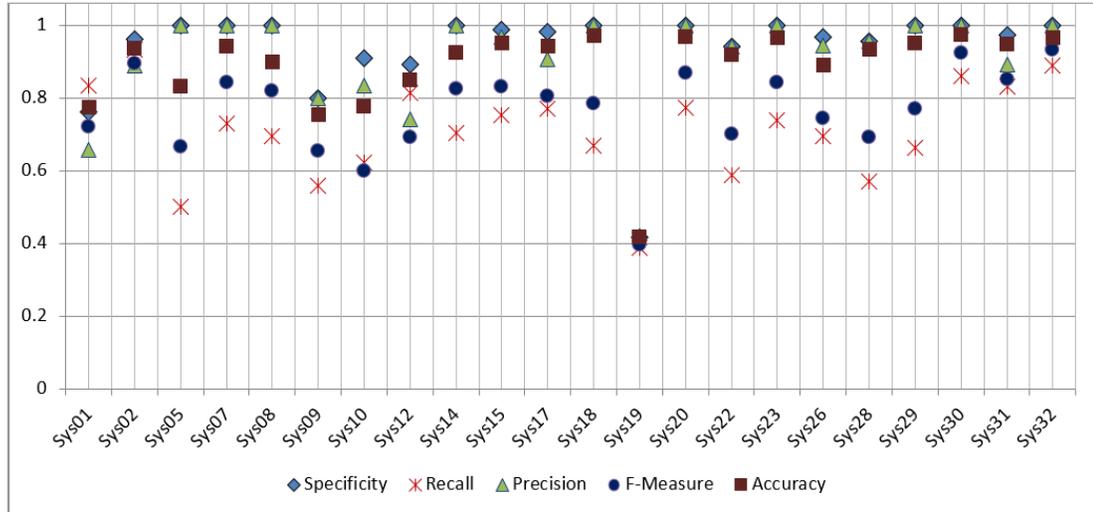


Figure 22: Rules Performance - Learning systems one by one

Figure 23 shows the percentage of times the rules have been considered. Our aim here is to provide a kind of score of each rule that assists in selecting the rules in the future when apply the rules on real applications that have no the actual target model. The scores can be used to rank the rules then we apply the rules on the validation systems based on their ranking. For instance, *Rule<4>* which has a high score 0.77 will be considered first when evaluating the rules against the validation systems or even in future application.

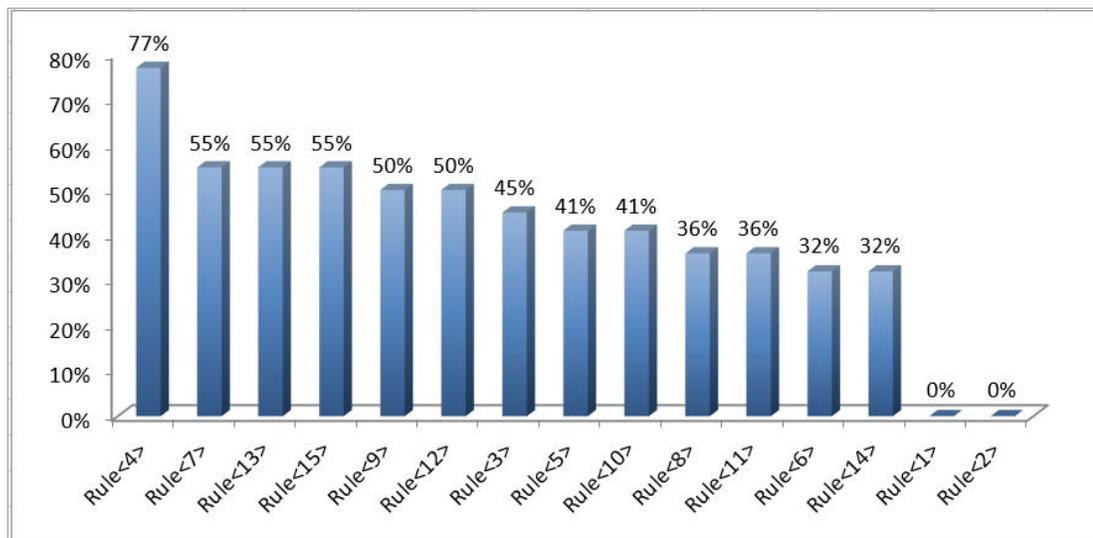


Figure 23: Scores assigned to each rule based on their performance.

6.4.2 Validation of Induced Rules

The best final rules resulting from the learning phase have been validated in this experiment against the set of validation samples. Starting from the top ranked two rules then the next ranked rule was added one by one. Figure 24 demonstrates the overall average of accuracy measures resulting from validation using 12 samples with different number of rules. It is obvious considering 3, 4 or 5 best rules gives similar results. Increasing or decreasing the number of rules varies from one system to another. For example, the systems 6, 13, 25 and 34 have a steady accuracy measures starting from 4 till 15 rules. In case of systems 3, 4 and 23, the accuracy measures decreased when adding more rules. On the other hand, the accuracy measures were improved when adding more rules in case of systems 11, 16, 24 and 33. These occurred changes are very low thus when calculating the average we get close values.

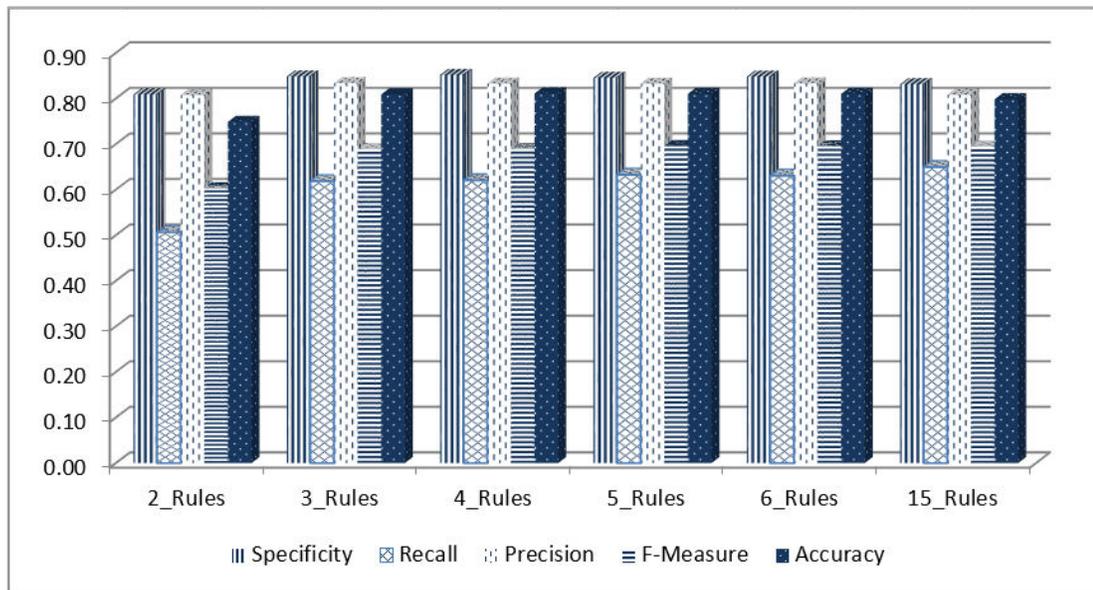


Figure 24: Average of accuracy measures (validation systems) with different number of rules.

It is obvious from Figure 24, the combination of the 15 rules gives a performance which is less than using 6 or 5 rules. Indeed, the rules here are ranked based on their performance and application time. Thus, it is reasonable when adding more rules some of them may have a negative impact on the other rules. When adding more rules it may result in grouping more classes in incorrect packages. The results of another selection method we considered are shown in Figure 25. We selected three different sets of rules randomly and measured the performance for each set individually. Each set consists of 7 rules as follows: subset1 = { 1, 3, 4, 8, 9, 12, 16 }, subset2 = { 2, 3, 4, 7, 9, 14, 16 } and subset3 = { 3, 4, 7, 8, 10, 12, 13 }.

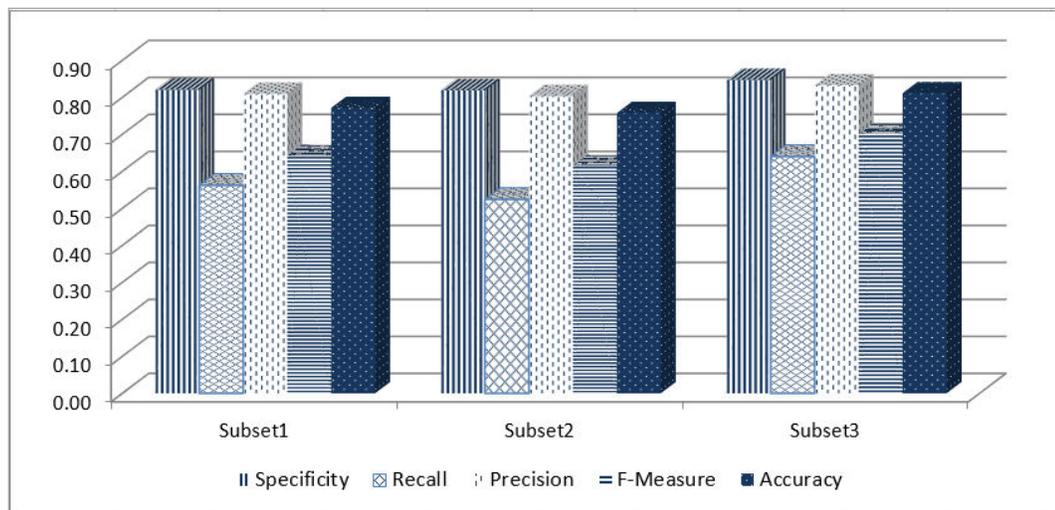


Figure 25: Average of accuracy measures (validation systems) with random selected rules.

6.4.3 Accuracy of Façades Rules

For the task of introducing façade interface only 13 systems, that use this practice, have been used for learning and validation in the ratio 2:1. The learning systems present different forms of using Façade. Nevertheless, ALEPH generalizes only one rule for all examples.

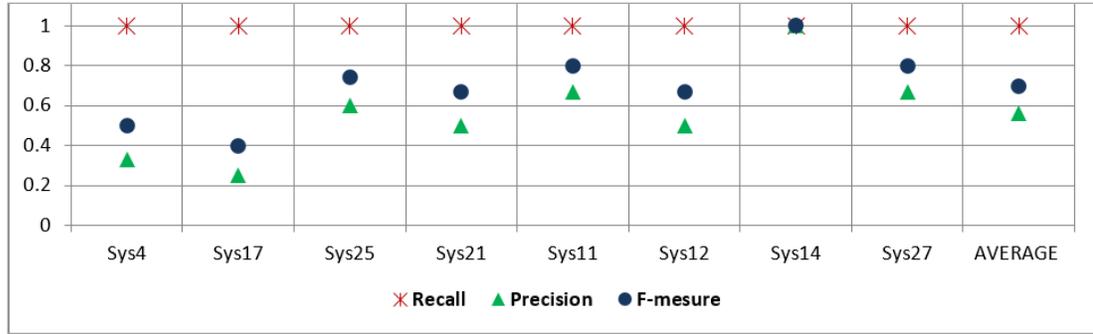


Figure 26: Accuracy measures - Learning Systems

When ALEPH generalizes the target clause, it looks for the minimal number of atoms that can cover the given examples. When generalizing the given learning examples, the learner considers only the type of relations not the count. Thus the problem is seen like this; when a package p has an external relation linked to one of its classes, add a façade to the package p .

Since when having only one rule we applied the induced rule directly on the learning and validation samples one by one. Figure 26 shows the accuracy measures when applying the induced unique rule on the eight learning systems. Only three measurements used for this experiment because there are true negatives can be collected here.

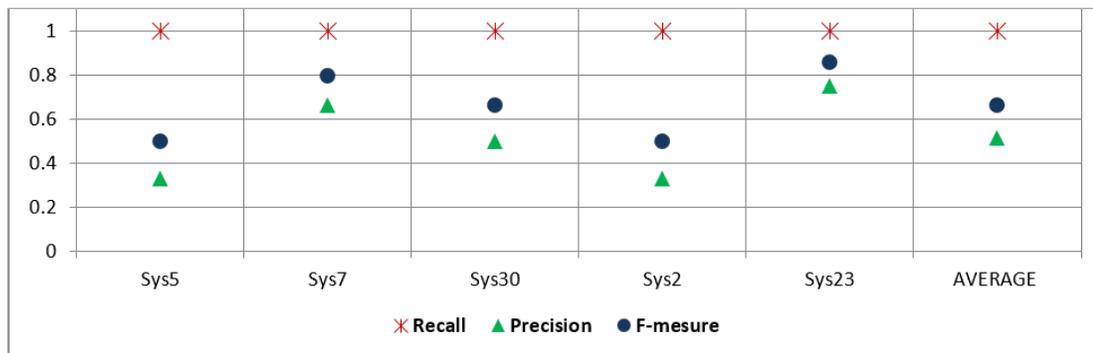


Figure 27: Accuracy Measures - Validation Results

In the same way, the induced rule has been applied on the validation systems. Figure 27 demonstrates the accuracy measures when applying the rules on the validation systems.

It is obvious, application of the rules results in a full recall for all systems. This is because the rule introduces a façade interface for each package. That is, it guarantees introducing all the interfaces presented in the actual target models. However, it introduces more interfaces which impact highly the precision and consequently the final f-measure.

6.5 Discussion

Although ALEPH has been used widely in the literature, the induced hypotheses in the tackled problems have small arity in their head predicates. For instance, the arity of *packageOfClasses(X,Y)* is two. ALEPH requires to specify each argument type and whether it is *input (+)* or *output (-)* as used in Table 10. The types should be maintained also in the body predicates. In our context, the arity of *packageOfClasses* changes based on the number of classes located on the corresponding package. Thus, there is a need to adjust the used modes and types in each run. This caused a problem when having a large arity. During rules induction phase, we noticed that when providing examples of packages having five classes or more, it was not possible to generalize hypotheses for such examples.

Another observation is that, ALEPH needs at least two similar examples to generalize a hypothesis. If no similar patterns have been seen in the given examples while training, it will not be possible to synthesize the right output. In reality, for one example having a large arity, the opportunity to find another example having the same number and type of relations is low. On the other hand for the examples consisting of two/three classes, all

the examples have been covered. Inversely, the opportunity to find a similar example is better where the possible relations among the classes are limited.

Moreover, it is noticeable the measures presented in Figure 22 vary from one system to another. The reason behind that is the nature of the used examples to generate the transformation rules and also the ability of ALEPH to generalize the given examples. For example, the performance in case of Sys 19, presented in the learning, was the worst. This system has 28 classes placed in 4 packages. When learning the rules it was not possible to learn such rules as explained above. On the other hand, the relations among the classes are not easily to be covered by the already induced rules. In other words, although this system has been presented in the learning, the induced rules by ALEPH could not cover such example.

Another observation, the number of the rule-created interfaces in the different learning and validation systems is either equal or more than the number of interfaces in the actual design. Thus, we get a full recall in almost all the cases, shown in Figure 27. The reason behind that is that the rule will introduce an interface between two packages whenever there is a relation between their classes.

6.5.1 Open Issues

In this section we discuss some open issues related to the usage of ALEPH system to derive source-target transformation rules. We also identify preliminary possible solutions and corresponding future research tasks.

Our experiments revealed that when two artifacts have more than one relation of the same type (e.g. association), ALEPH induces a rule that considers only the type of the rule

regardless of the number of instances. That is, when two artifacts (packages or classes) have two (or more) associations connecting each other, ALEPH shows only the type of the relation not their counts. This has an impact on the generated mappings since the number of associations among a set of surely impacts corresponding design decisions.

A simple example of this shortcoming is shown in Figure 28 to give a glance. In the figure, the source model part shows four relations linking “*Package 1*” to “*Package 2*”. Based on these relations, a facade has been introduced shown in the target model part. However the induced rule by ALEPH considers only the type of the relation and ignores the count of the relations. Clearly, as manifested in this example, the count is important factor for introducing facades.

ALEPH system uses the given background knowledge along with the given examples to learn (i.e., generalize) rules. Thus, it expects more than one positive example to learn the rule, otherwise it returns the unique positive example as it is (i.e., without induction of rules). However, occasionally, generating a rule from just one example might be desirable for future improvement as more examples emerge, as with the case of incremental learning.

ALEPH and almost all the current ILP systems enforce declaring the modes and determination statements of the predicates supposed to appear in any clause hypothesized by the ILP system. That is, when using the system it is supposed to prepare the appropriate modes for each target hypothesis. For instance, by using the declaration presented in Table 10 we use “*packageOfClasses*” as the head of the target hypothesis and inheritance as the body. In some situations, we have no idea about the type of

relation. Therefore, specifying the modes in advance is impractical when there are clear relations between the head and the body of the hypothesis.

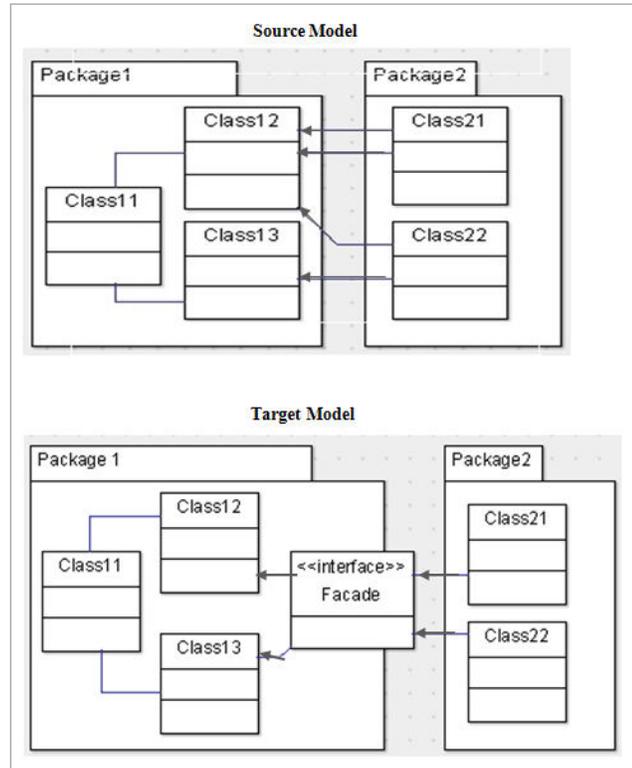


Figure 28: Example to show the drawback of the derived rule

6.5.2 The Current ILP Systems in MDD Context

ILP methods seem more suitable for inducing transformation rules. In addition, ALEPH has been applied effectively in various domains.

During our initial attempts to use ALEPH system, we used another ILP system called GILPS (General Inductive Logic Programming System) [151]. GILPS is a general ILP framework that implements various operations: Top-directed declarative search bias, a bottom-up ILP learner called *ProGolem*, and an efficient θ -subsumption engine called *Subsumer*.

Nevertheless, we didn't obtain promising results i.e. transformation rules of requirement analysis into software design. We recorded a number of limitations when applying the aforementioned systems [28]. These limitations related to the resultant rules, due to the strategy of induction used by these systems.

The power of ILP techniques can be distinguished by their expressiveness due to using logical descriptions as well as the ability to learn relational concepts. Nevertheless, currently most well-known and successful ILP systems make the following assumptions that may restrict their application.

- First, most techniques assume that explicit negative examples of the target predicate are available or can be computed using CWA, but for some problems like the problem we aim to tackle explicit negative examples are not available. Furthermore, using CWA to compute a set of negative examples may result in large or infinite set.
- Second assumption is that the resultant logic programs are expressed in pure Prolog and most of ILP systems employ the Prolog programming language. Prolog has its own limitations when applied to software models due to the nature of the models and many other points as discussed in [29].
- Third, most techniques enforce declaring the modes of call for predicates that can appear in any clause hypothesized by the ILP system. That is, when using the system it is supposed to prepare the appropriate modes for each target hypothesis, which is impractical when there are clear relations between the head and the body of the hypothesis.

Due to these limitations, and the nature of the relations linking the analysis and design artifacts we were unable to get reasonable results on learning analysis-design transformation rules by employing ALEPH or GILPS systems.

Using ALEPH in the different applications showed promising results, however it did not perform adequately on the problem investigated. The obtained results do not reflect perfectly the relations between the analysis and design artifacts. These limitations are discussed in the following.

- Even though the transformation problem has multiple predicates to be learned, the current ILP systems, such as ALEPH and GILPS, deal with the problem as a single predicate learning problem. That is each predicate will be learned independently in a separated run regardless of the resultant hypotheses from other runs.
- ALEPH algorithm starts by selecting one of positive examples, then builds the most-specific clause entailed by the example. After that, it follows different search strategies for generalization and reduction. The covering approach used does not assist in reflecting the real relations between the analysis and design artifacts. For instance, Figure 28 presents a rule that introduces an interface to a package. Indeed, in the provided training examples at least there are three associations' relations linking the two packages to introduce an interface. However, because of the reduction step, the algorithm only considers one predicate and it ignores the predicate's occurrences count. It is important to mention that comparable results were obtained when using GILPS system.

- ALEPH, GILPS and almost all the current ILP systems enforce declaring the modes of call for predicates that can appear in any clause hypothesized by the ILP system. That is, when using the system it is supposed to prepare the appropriate modes for each target hypothesis, which is impractical when there are clear relations between the head and the body of the hypothesis.
- Occasionally in the software design model particular artifact may appear only once, i.e. for this artifact only one positive example is presented. In this case, the used ILP systems are not able to learn the relationship between the unique positive example and the other atoms appearing in the background knowledge. Although learning the relation and inducing the rule based on one positive example might end up with imperfect rule, it is essential in some cases. The confidence of the resultant rule is low; however there are some situations where the software models may present only one example.

Nevertheless, overcoming the aforementioned limitations may result in a promising and trusting approach to enrich the proposed transformation system. It is possible to implement learning algorithm in any language other than PROLOG. The ILP power in inductive learning from relational descriptions and its precious and formal expressiveness can be employed to build a system to derive the intended transformation rules. Accordingly, in the next chapter we present a new ILP system to induce MDD transformation rules with overcoming the presented limitations.

CHAPTER 7

THE PROPOSED ILP SYSTEM

In Chapter 5, we described the model transformation environment proposed to achieve the transformation by examples. Most of the presented components, such as preprocessing, rules application and evaluation, are proposed specifically for MDD transformations by examples, which is one of the contributions of this work. Chapter 6 presented the application of ALEPH system through the proposed model transformation system. In this chapter, we present an overview, fundamental concepts, justifications, and algorithms of a new incremental ILP system, called *Model Transformation using Inductive Logic Programming (MTILP)*.

7.1 An overview

We proposed this system to overcome the limitation recorded when employing ALEPH system in MDD transformations. The MTILP system is not a replacement for ALEPH, it can be used to solve problems that couldn't be tackled by ALEPH system.

In the same way we employed ALPEH system to induce transformation rules, MTILP system is employed. Furthermore, in the proposed system it is possible to batch the background knowledge and all the positive examples one time to induce all the rules. In contrast, ALEPH needs more effort to feed different files at each run where the given positive and negative examples should have the same predicate and arity.

In general, ILP was originally designed to deal with binary classification tasks. This inductive learner generalizes the given training examples through the identification of the features that can be used to empirically differentiate positive from negative training examples. Hence, ILP generally requires a number of positive and negative training examples. However, there are some application domains where examples do not exist, or are scarce. For example, learning the transformation rules is not concerned with distinguishing positive examples from negative ones. It is only concerned with the positive examples. This kind of situation has driven the development of positive-only learning [148]. In our system, inducing the hypotheses can be achieved using the positive examples only. However, if the negative examples are available they can be used to reduce the search space and to come up with consistent rules in the generalization process.

As we have already seen in Chapter 6 existing ILP systems are using predicate and mode declarations as a language bias in order to make the ILP learning algorithm more efficient. In our setting, we also impose some language bias. To be more specific, we do introduce the following kind of options.

- The system allows the user to specify the relevance level. That is the hypothesis construction starts by searching through the background knowledge for the related atoms. The relevance level allows consider/ignore the indirect related atoms.
- The declarations of modes that specify the structure of the intended hypotheses is optional. The learner can utilize the given predicates (as modes) or it can extract

the head and body of the hypothesis by using the positive examples and the related atoms provided in the background knowledge.

- MTILP allows to start learning by using positive examples along with the background knowledge. In some cases, the negatives examples are not available to be used for learning as in the context of MDD transformation. Thus using negative examples is an option to help pruning large parts of the search space.
- In the MDD transformation problem, although no negative examples can be provided, MTILP provides another way for learning from the negative examples. After inducing the rules using the positives only, the rules are applied. After that, if there are negative examples resulted from applying the induced rules then the learning can be repeated using both examples.

7.2 MTILP Generic Algorithm

The flowchart in Figure 29 describes the MTILP algorithm. The algorithm starts by taking the background facts and the examples from the repository. Three or two files are expected to be used as input based on the availability of the negative examples. It then checks the groups of the positive examples. It is important to pay attention for the possible dependence among the different hypotheses. If the groups of the given examples have the same predicate, regardless of the arity, then the order is not significant factor. On the other hand, if the groups of the given examples have different predicates, then in many cases there is a need for a particular order to start learning.

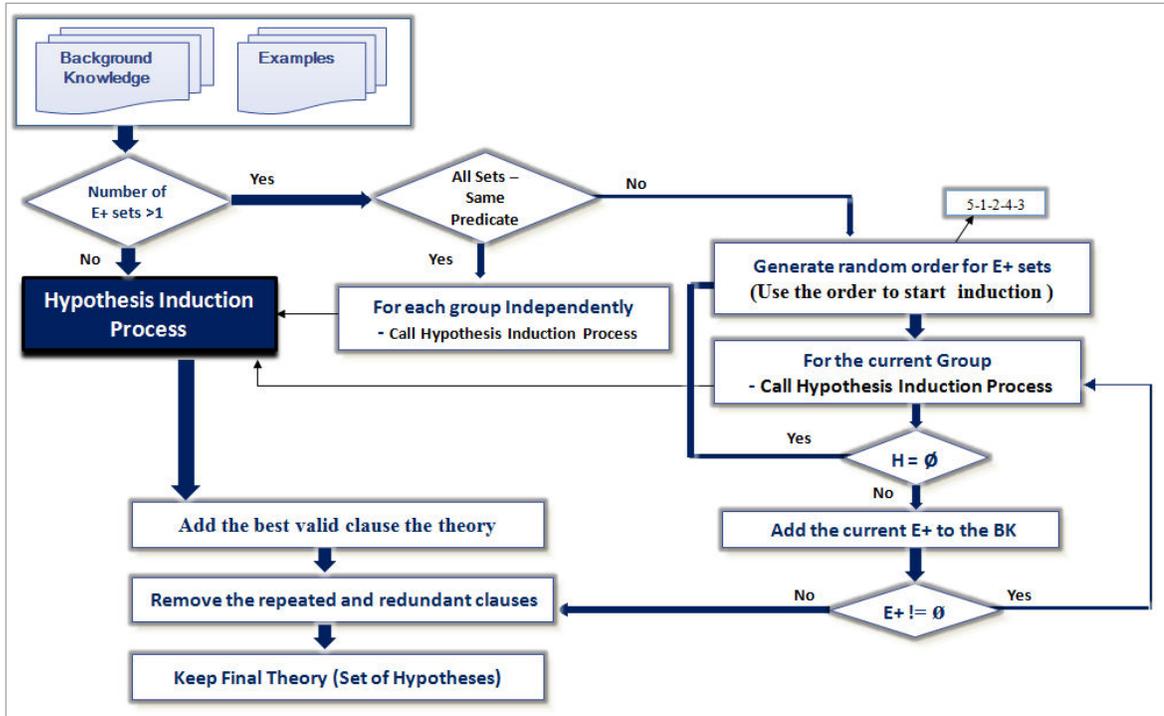


Figure 29: Flowchart of MTILP Algorithm

It was explained in Chapter 5 and Chapter 6, the given examples are grouped based on two criteria: the predicate name and the arity. Thus, it is expected to have one or more sets, each having a number of positive examples that share the same predicate and the number of arguments. For instance, assume the examples are grouped as follows:

$$E\text{-set1: } \{p(x,y,z), p(a,b,c), p(w,v,y)\}$$

$$E\text{-set2: } \{p(m,n), p(k,l)\}$$

$$E\text{-set3: } \{q(o, p, t), q(h, f, s)\}$$

MTILP starts by checking the number of sets. In case only one set is provided, it goes directly to the induction process (detailed in the following). If the given examples contain more than one set, it goes further to check the used predicate for each set. For instance, in case the given sets are *E-set1* and *E-set2*. Although the two sets have different arity (three

and two respectively), they use the same predicate p . In this case, the order of learning doesn't cause contradiction. However, it needs to go further if the sets have different predicates and consider the incremental learning approach.

7.2.1 Incremental Learning

We refer to incremental learning when there is a need to learn predicates that might have dependency. In some cases, it is not possible to learn one predicate before a particular one. For instance, when learning the rules to introduce Façade, we noticed that three different artifacts would be added (façade interface, association from classes to façade, and from façade to classes). In this case, introducing façade artifacts should be learned first to be able to learn the two others. The user can specify the order of this kind of learning when there is enough knowledge about the dependency details. However, in many cases the user has a huge amount of data and wants to learn the rules without digging into the nature of relations.

When getting the three sets shown above $E\text{-set1}$, $E\text{-set2}$ and $E\text{-set3}$, the question is which one to select first for learning. MTILP initializes a random order for learning the predicates, say $\{2,1,3\}$. Then it calls the induction function to learn the target relation of $E\text{-set2}$ first. The measure here is to terminate and find another order in case the problem cannot be solved, i.e. if no hypothesis could be induced for a particular set.

In such learning, after inducing the hypothesis of a particular set, say $E\text{-set2}$, the positive examples $E\text{-set2}: \{p(m,n), p(k,l)\}$ are added to the background knowledge to be utilized in the next learning iteration. This process is repeated for all the given examples sets.

7.3 Hypothesis Induction Process

This process is the central part for inducing the hypothesis of the given examples. It is supposed to feed the background facts and examples set that have the same predicate and arity. MTILP differs from ALEPH and many other ILP systems in the way the examples are covered. While other systems use a covering approach, MTILP employs brute-force and genetic algorithms methods to find the possible combinations among the different examples.

Figure 30 demonstrates the whole induction process. Different phases have been considered when inducing a hypothesis using the examples and background knowledge. In the following we detail the implemented phases.

7.3.1 Hypothesis Space Construction

As discussed early, ILP system is a search problem. In such a problem, there are a number of candidate solutions to be searched. These solutions represent what is called the search space. To specify the *search space*, the generalization process starts by constructing the bottom clauses for all given positive examples.

Thus, the learning of the ILP system starts by the construction of the hypothesis space. An important concept in ILP is that of the most-specific clause, often called bottom clause. We can differentiate between two versions of the bottom clauses: ground and substituted bottom clauses.

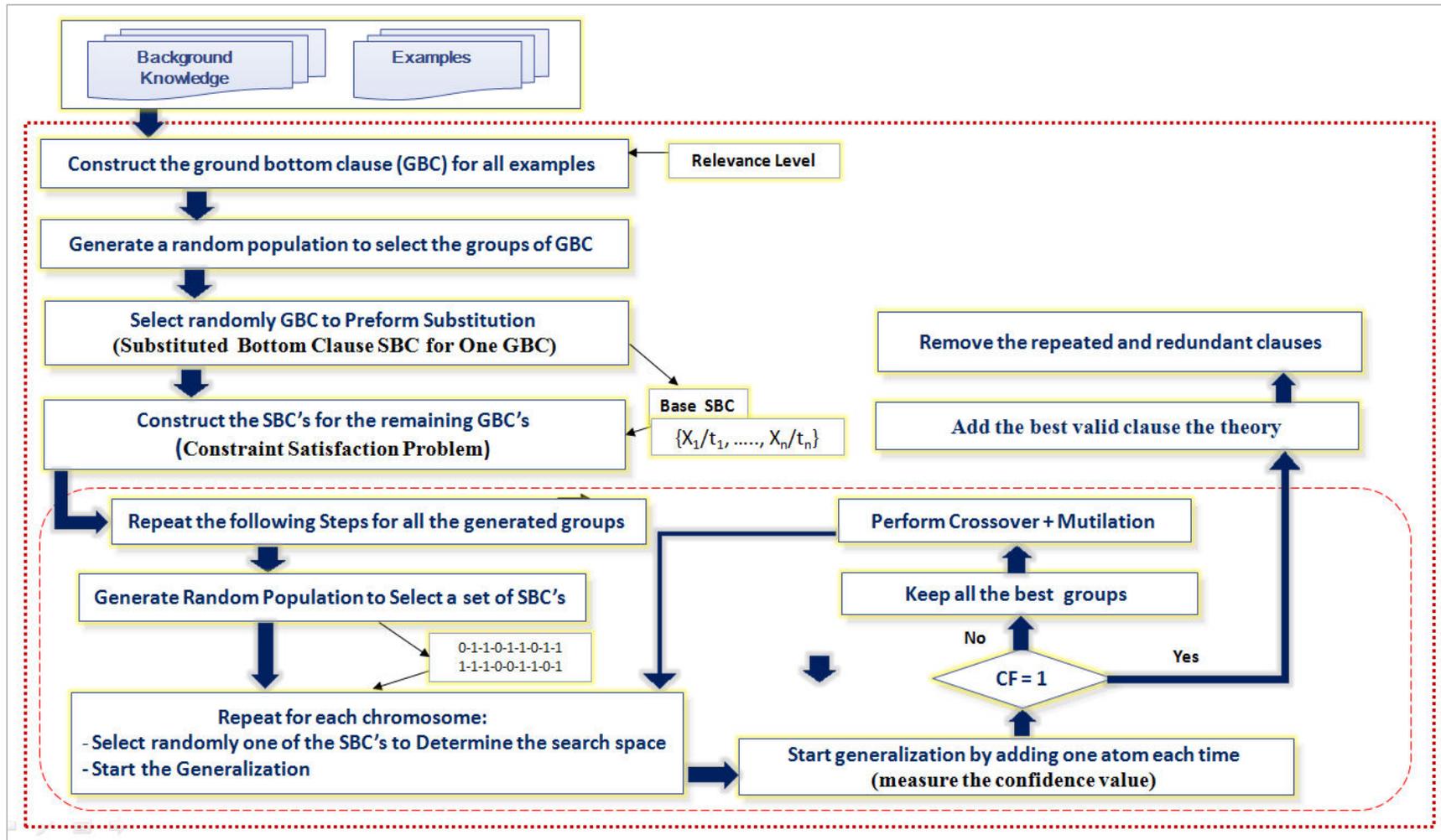


Figure 30: The Induction Process Flowchart

In the following, we consider two steps to construct the two types of bottom clauses. The first step is used for constructing the ground bottom clause (gbc), while the second step is to construct the substituted bottom clause (sbc).

7.3.2 Bottom Clauses Construction

For each positive example, e_i^+ , a ground bottom clause gbc_i is constructed ($\forall e^+ \in E^+ \Rightarrow gbc$) where gbc_i contains all facts known to be true about the current example e_i^+ .

For each e_i^+ , it is supposed to find all the related atoms in the background file. As a result, we end up with a number of bottom clause equals to the number of the positives presented in the current group. The construction of gbc is determined by the language bias, if there is any. The given bias describes some restrictions to be considered in order to build the target bottom clause.

Algorithm 4 describes the steps followed to find the bottom clauses for all the given examples. It is supposed to consider only one E^+ group by an iteration. First, it takes the examples one by one. For a particular example $e^+ \in E^+$, the algorithm searches for all related atoms existing in the background knowledge. The atom is related directly to e^+ when one of its arguments appears in e^+ . Other indirect related atoms can be added based on the setting of the relevance level (explained in the following).

Consequently, each iteration results in bottom clauses equal to the number of positive examples presented in each group.

Algorithm 4: Bottom Clauses Construction

Input: Background knowledge K ,
groups of positive examples $E = \{gE_1, gE_2, \dots, gE_n\}$,
user-defined level of relevance L ($1 \leq L \leq 3$).

Output: gbc for each $e^+ \in E$

- 1: Let $GBC = \{\}$
- 2: **for** $i = 1 : n$ (number of E groups) **do**
- 3: Let $gbc_i \leftarrow \{\}$
- 4: **for** $j = 1$: number of examples (e_j^+) in gE_i
- 5: $gbc_{ij} \leftarrow \{\}$
- 6: Let gbc_j head= e_j^+
- 7: **for** each argument constant in e_j^+ **do**
- 8: search for the occurrence of the current argument in K
- 9: for the ground fact f contains e_j^+ argument add f to the body of gbc_j
- 10: **end for**
- 11: **for** $a = 2 : L$ (level of relevance)
- 12: from gbc_{ij} body atoms find all arguments $args$ not in gbc_{ij} head
- 13: search for the occurrence of each found $args$ in K
- 14: gbc_{ij} body+= fact f contains any $args$
- 15: **end for** a
- 16: $gbc_{ij} := gbc_{ij} \cup gbc_i$
- 17: **end for** j
- 18: $GBC := GBC \cup gbc_i$
- 19: **end for** i
- 20: **return** GBC .

When having a target instance, $pred(arg_1, arg_2, \dots, arg_n)$, the related atoms are searched in the background knowledge. An atom a_i is a *direct related atom* to $pred(arg_1, arg_2, \dots, arg_n)$ if it shares a constant argument arg_i appearing in the target instance. For example, given the target instance, $a(x, y)$, and the background $facts = \{b(x, z), c(z, k, l), d(y, m)\}$. The facts $b(x, z)$ and $d(y, m)$ are considered direct related atoms.

In contrast to direct related atoms, an atom a_i is an *indirect related atom* to $pred(arg_1, arg_2, \dots, arg_n)$, if doesn't share any argument arg_i with the given target instance. In the previous examples the fact $c(z, k, l)$ is seen as indirect related to the target instance, $a(x, y)$.

However, the type of indirect related atoms might be involved in the bottom clause. Thus, our algorithm provides a parameter, called *relevance level*. This parameter can be adjusted by the user to specify the desire to involve the indirect related atoms in the construction process. The level of relevance (L) can get one of three values 1, 2, or 3. With $L = 1$ only the literals having, as input variables, input variables of the head (layer 0) are added to the most-specific clause. At layer i only literals having input variables appearing in layer $i - 1$ (as output or input variables) can be constructed. It is important to note that with a low value for L not all facts from the background knowledge will appear in a most-specific clause.

To illustrate the impact of L, let us consider the illustrative example presented in Section 2.3.1.1. The ground most-specific clause of the same example *daughter/2*, when $L=1$ is demonstrated Figure 31. It is obvious that only the variables appearing in the body of the bottom clause are used as input (the head of the bottom clause).

```
daughter(mary,ann)←  
parent(ann,mary), female(mary),  
parent(ann,tom), female(ann).
```

Figure 31: Ground most-specific clause for example *daughter /2* when $L = 1$

In contrast when constructing *gbc* for the same example *daughter /2* with using $L = 3$, the direct and indirect atoms are involved as shown in Figure 32.

Thus, it is important to utilize a user-defined parameter L when constructing the most-specific clause to specify the number of levels of variables to consider. The choice of the value for L specifies the hypothesis space size. With a low value for L the target concept may not be present in the hypothesis space as the required atoms may not occur in the most-specific clause. The usage of the relevance level value is described in Algorithm 5.

```
daughter(mary,ann)←
  parent(ann,mary), female(mary),
  parent(ann,tom), female(ann).
  parent(tom,eve), female(eve) .
```

Figure 32: Ground most-specific clause for example *daughter/2* when $L = 3$

7.3.3 Bottom Clauses Substitution

A substitution is defined as the operation that replaces variables occurring in each *gbc* by terms (values). For example, in Figure 31 we could replace the variable *mary* by the term X . A substitution θ can be defined as a finite set of the form: $\{x_1/t_1, \dots, x_n/t_n\}, n \geq 0$, where the x_i are distinct variables and the t_i are terms, i.e. t_i is substituted for x_i .

In this regard, we use the alphabet letters $\{A, B, C, \dots\}$ to achieve the substitution operation. Given a set of the ground most-specific clauses resulting from the previous operation. It is supposed to replace each variable of the *gbc* by a unique value. Figure 33 demonstrates a substituted bottom clause resulting from applying substitution on the *gbc*

presented in Figure 31. In the sequel, when we refer to the substituted bottom clause as bottom clause and use a symbol \perp , to refer to it. We aim to apply this substitution to make a particular gbc_i more specific in order to be matched with another gbc_j . To find the suitable substitution for all the ground bottom clauses, we solve it as a constraint satisfaction problem (CSP [114]). Figure 33 shows the substituted bottom clause for the ground bottom clause shown in Figure 31.

*daughter(A,B)←
parent(B,A), female(A),
parent(B,C), female(B).*

Figure 33: A substituted most-specific clause

The problem can be expressed in the following form; given a set of variables $\{x_1, x_2, \dots, x_n\}$ and a finite set of possible terms $\{t_1, t_2, \dots, t_p\}$ that can be assigned to variable x_i , and set of constraints $\{y_1, y_2, \dots, y_m\}$.

The constraint y_i involves subset of the variables and identifies the possible combinations of terms for that subset. That is, the problem can be defined by an assignment of terms to some or all of the variables such that $\{x_i = t_i, x_j = t_j, \dots\}$ with satisfying constraints.

Algorithm 5 demonstrates the steps followed to achieve the appropriate substitution. The aim is to apply a particular substitution to a set of expressions (values or bottom clauses) to make them identical. In first-order logic, the unification problem can be expressed as follow: given two terms containing some variables, find if there is a substitution (assignment of some variable to every value) that makes the two variables equal.

Input: GBC and terms $T = \{A, B, C, \dots\}$

Output: Substituted Bottom Clauses for each group

```
1: Let  $SBC \leftarrow \{\}$ 
2: for  $i=1$  : GBC groups ( $gbc_i$ ) do
3:   Let  $sbc_i \leftarrow \{\}$ 
4:   select randomly  $gbc_{i_r}$ 
5:   repeat for each argument  $g$  in  $gbc_{i_r}$ .head
6:     substitute  $g_k$  in  $gbc_{i_r}$ .head with a term  $t_k$  from  $T$ 
7:     search for the occurrences of  $g_k$  in the  $gbc_{i_r}$ .body
8:     substitute  $g_k$  with  $t_k$ 
9:   end repeat
10:   $sbc_i := sbc_i \cup sbc_{i_r}$ 
11:  use  $sbc_{i_r}$  as the set of constraints
12:  repeat for each remaining  $gbc_{i_j}$ 
13:     $sbc_{i_j} =$  call substitute_with_constraints( $gbc_{i_j}$ ,  $T$ ,  $sbc_{i_r}$ )
14:     $sbc_i := sbc_i \cup sbc_{i_j}$ 
15:  end repeat
16:   $SBC := SBC \cup sbc_i$ 
17: end for  $i$ 
18: return  $SBC$ 
```

function `substitute_with_constraints` (s set of variables V ,
a set of terms T , and a set of constraints C)

```
19:  $h \leftarrow$  number of  $V$  head arguments  $g_k$ 
20:  $F =$  a subset of  $T$  consists of the first  $n$  terms
21: find a permutation  $\rho_i$  of  $F$  elements
22: substitute head argument  $g_k$  with a term  $\tau_k$ 
23: search for the occurrences of  $g_k$  in  $V$  body
24: substitute  $g_k$  with  $\tau_k$ 
25: measure  $C$  satisfaction for  $\rho_i$ 
26: if not satisfied call substitute_with_constraints.
27: consider best permutation satisfying  $C$ 
28:  $var \leftarrow$  select unsubstituted variables in  $V$ 
29: for each arguments  $g_k$  in  $var$ 
30: substitute  $g_k$  with a next term from  $T$ 
31: return  $sbc$ 
```

For a particular given ground bottom clauses set, it starts randomly one of the clauses gbc_r to perform a direct substitution for all its variables (i.e. with no constraints). Then the resulting substituted bottom clause \perp is used as the set of constraints that should be satisfied when applying substitution on the remaining gbc_i belong to the current set. The algorithm follows the backtracking search strategy, where it selects term for one variable at a time and backtracks when a variable with the assigned term doesn't satisfy the constraints. In our transformation problem, as stated earlier, building the gbc results in a number of ground bottom clauses equals the number of positive examples. This can be formally expressed as follows. Let $\theta = \{x_1/t_1, \dots, x_n/t_n\}, n \geq 0$ be a substitution, and P an expression. Then $P\theta$, the instance of P by θ , is the expression obtained from P by simultaneously replacing each occurrence of x_i by t_i .

7.4 Search for the Hypothesis

Before starting the search of the candidate hypotheses, different combinations of the resulting bottom clauses are generated. According to their number, one of two approaches can be used to find the possible combinations. One approach is a brute-force (BF) to find all possible combinations when dealing with a small number. Another approach is a genetic-algorithm (GA) to generate a random population representing particular combinations, and then several generations can be found.

Each combination is considered individually to perform clauses generalization and to learn the expected hypotheses. After that, from each presented set of bottom clauses, a random bottom clause is selected first to determine the hypothesis space. The combination operation results in several sets of bottom clauses. The concern here is to

generalize one hypothesis in each set to include a number of the bottom clauses in the specified set. In turn, it covers the corresponding positive examples. In each set all the bottom clauses share the same head, thus one of them will be used as the head of the generalized rule. Then in the same set we search for the commonalities of the atoms among the different bottom clauses. It is expected to induce one or more rules from each set. Algorithm 6 demonstrates the steps followed to accomplish this operation.

The rationale behind considering several combinations for induction is to give another view about the given examples. Thus, from each combination set, either using GA or BF, one bottom clause \perp is selected randomly to be used to determine the hypothesis space.

The head of the selected \perp is used as the head of the candidate hypothesis \hat{h} . Then each time, add one atom $l_i \in \perp$ to \hat{h} and measure the confidence value v with considering the other bottom clauses in the current set. According to v , l_i is kept in \hat{h} or retracted. The calculation of v considers the number of covered bottom clauses in the current set when adding l_i . By adding new atom each time, v can be improved or impacted. This represents the search operation to find the best hypothesis as a solution for the presented ILP problem. Figure 34 shows an explanation of this operation.

Algorithm 6: Bottom Clauses Generalization

Input: $SBC = (\{bc_{11}, \dots, bc_{1n}\}, (bc_{21}, \dots, bc_{2m}) \dots (bc_{k1}, \dots, bc_{ky}))$

Output: A set of induced hypotheses Σ

```
1: Let  $\lambda = 10$  (Threshold to use GA)
2: Let  $\Sigma := \{\}$ 
3: while  $SBC \neq \emptyset$ 
4:   Let  $BC_i$  one set of  $SBC$ 
5:   if  $(BC_i.size \geq \lambda)$ 
6:      $H_i \leftarrow$  call induction_by_GA ( $BC_i$ )
7:   else
8:      $H_i \leftarrow$  call induction_by_BF ( $BC_i$ )
9:   end if
10:   $\Sigma := \Sigma \cup H_i$ 
11:   $SBC := SBC \setminus BC_i$ 
12: end while
13: return  $\Sigma$ 

14: function induction_by_GA ( $A$  set  $BC = \{bc_1, bc_2, \dots, bc_n\}$ )
15: Let  $\hat{C}$  = Num of chromosomes (defined by the user)
16: Let  $B$  = Num of clauses in  $BC$ 
17: Let  $\hat{P}$  matrix of size  $[\hat{C}][B]$ , each cell 0 or 1 randomly
18: Let  $\hat{H} := \{\}$ 
19: repeat for each chromosome  $ch_i \in \hat{P}$ 
20:   Let  $\hat{S} := \{\}$ 
21:    $\hat{S} := \hat{S} \cup bc_j \in BC$  iff  $gen_j = 1$ , where  $gen_j \in ch_i$ 
22:   Select randomly  $bc \in \hat{S}$ 
23:   Let  $\hat{h}_i$  a candidate hypothesis s.t.
24:      $\hat{h}_i.head = bc.head$ ,  $\hat{h}_i \leftarrow \square$ 
25:   for each atom  $l_k \in bc.body$  do
26:     add  $\hat{h}_i \leftarrow \hat{h}_i \cup l_k$ 
27:     compute the confedence value  $\nu_i$  using all clauses  $\in BC$ 
28:     if  $\nu_i$  improves then consider  $l_k$ 
29:     otherwise backtrack to retract  $l_k$ 
30:   end for
31:   if  $\nu_i = 1$  (or threshold)
32:      $\hat{H} := \hat{H} \cup \hat{h}_i$ 
33: end repeat
34: return  $\hat{H}$ 

for the chromosomes not generalized
  Perform crossover and mutation
repeat setps 18 – 31
```

For each given example, MTILP traverses the search space using the refinement operator, adding/changing a new literal one at a time within the language constraints, keeping clauses with the best compressions at each refinement level along the way. All generated refinements must subsume the most specific clause under 0-subsumption (i.e. at least as general as the most specific clause). Using the lattice structure properties described in Figure 34, MTILP can calculate if none of the remaining candidate clauses nor their refinements will produce a better compression than the current best. When that happens, the best clause has been determined and the search can terminate. This process repeats until all given chromosome have been covered.

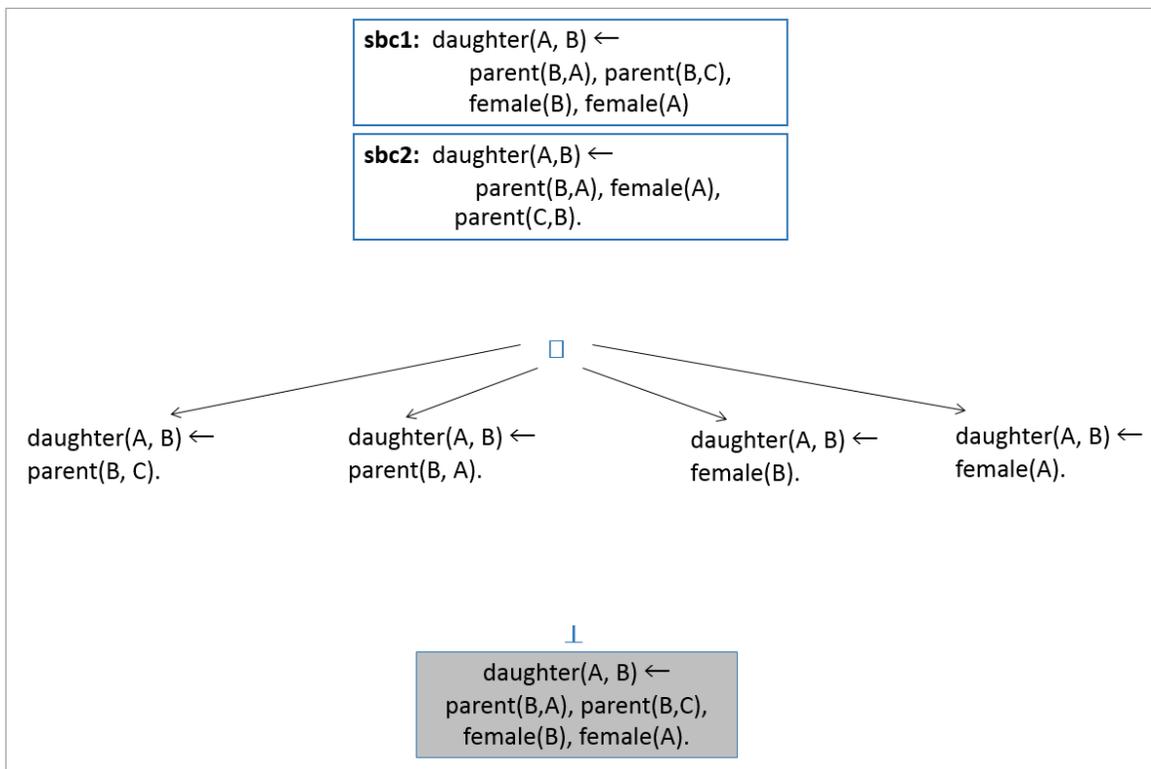


Figure 34: The construction of the search space when inducing daughter relation

For the *SBC*'s presented by each chromosome, it attempts to find the confidence value equals to 1 which is the maximum value. Figure 35 shows that MTILP ignores the atoms

that give a low confidence value. It determines the best confidence value for the second clause for instance ($daughter \leftarrow parent(B, A)$). The confidence value equals 1 because it the selected atom covers $sbc1$ and $sbc2$. Then MTILP goes further in the same direction seeking for more clauses that can cover the given SBC with keeping the max confidence value. It will follow the same procedure when adding more atoms as explained in the 2nd level. However, in the 3rd level when adding the available atoms, no one of them gives a full confidence value. Thus it ignores the 3rd level and consider the upper level with two atoms. Figure 36 shows the final induced hypothesis for the daughter relation using MTILP.

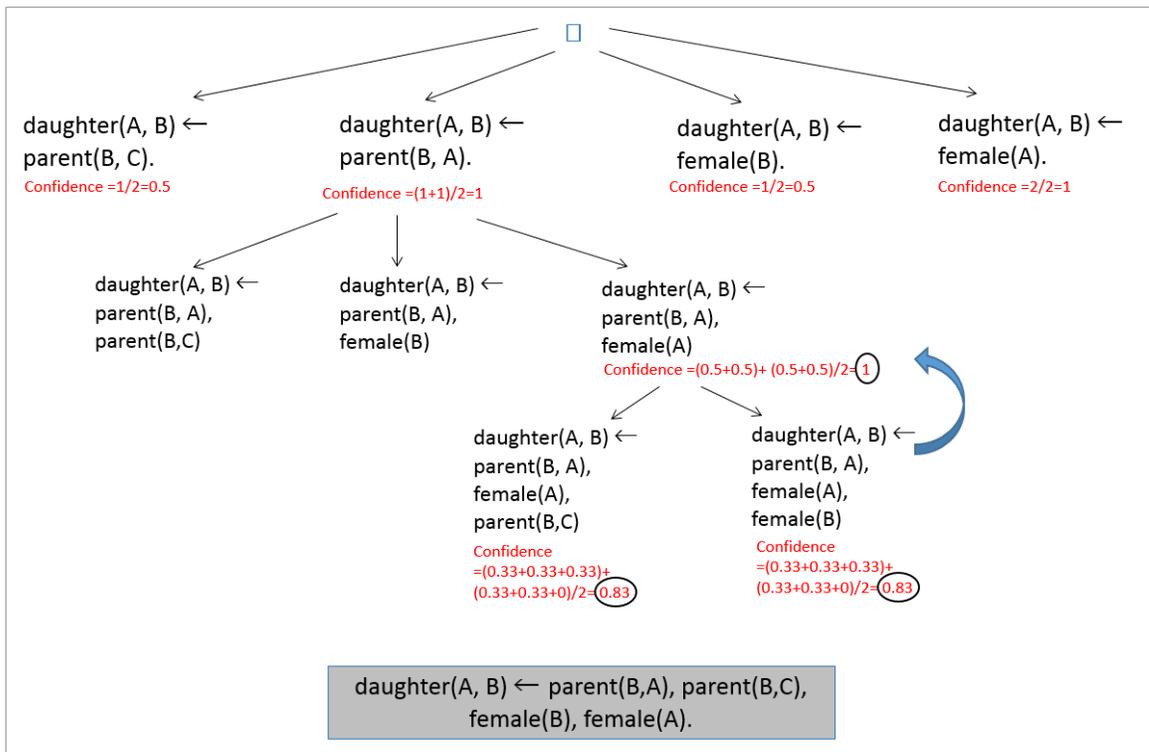


Figure 35: Snapshot of the search tree when inducing daughter relation

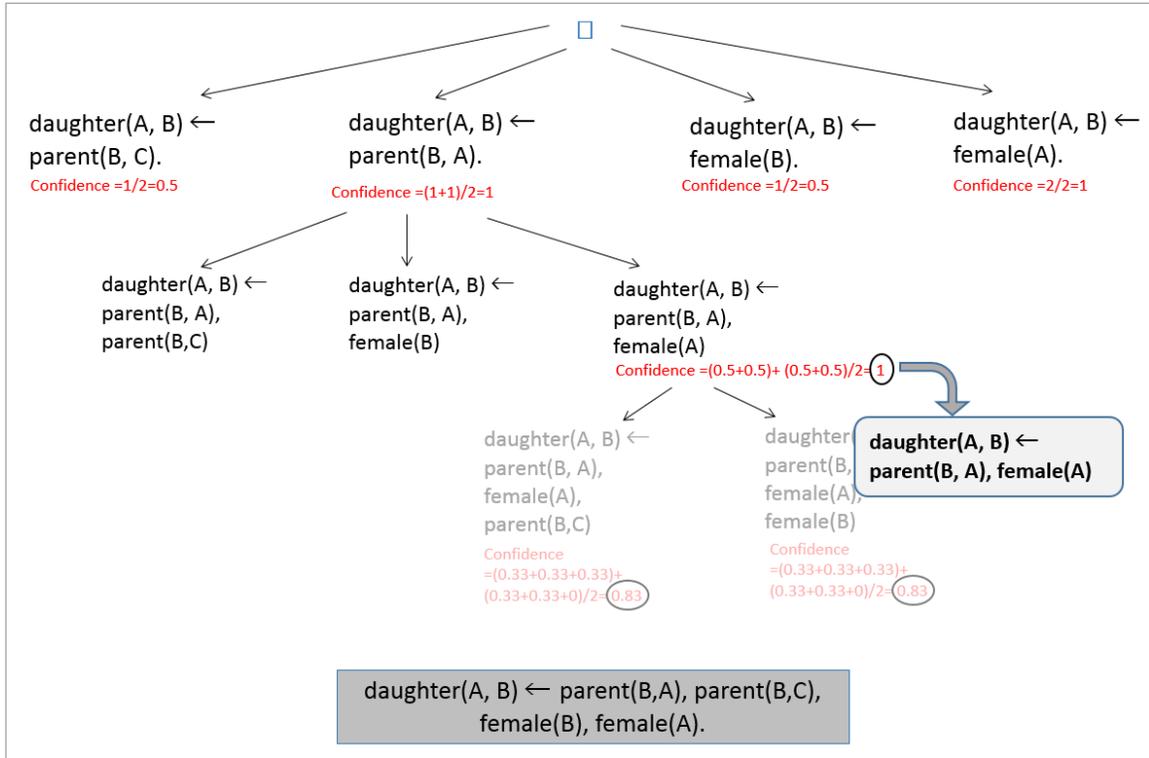


Figure 36: Snapshot of the final hypothesis using MTILP system

7.5 Learning from Negative Examples

The inductive learner was designed originally to solve the binary classification problems. Hence, the learner attempts to generalize the given examples by the finding the features distinguished empirically the positive from negative examples. As a result, there is a need to provide a number of positive and negative examples to start the induction. In addition, in the case of learning rules in general, the presence of negative data is considered one of the effective mean to prune the search space. Most of the induction systems require the use of negative data beside the positive ones to induce the needed hypotheses. However, critical problem of using negative examples since there are some domains where the negative data are not available, or are scarce. Such examples may not be available in

advance as we have seen in MDD context. That is, learning the transformation rules is not concerned with distinguishing positive examples from negative ones. It is only concerned with the positive examples. This kind of situation has driven the development of positive-only learning [148].

Based on that, the proposed system gives the user the opportunity to feed the negative data as input whenever they are available to be used during the search. If not the induction process can proceed further using the positive data.

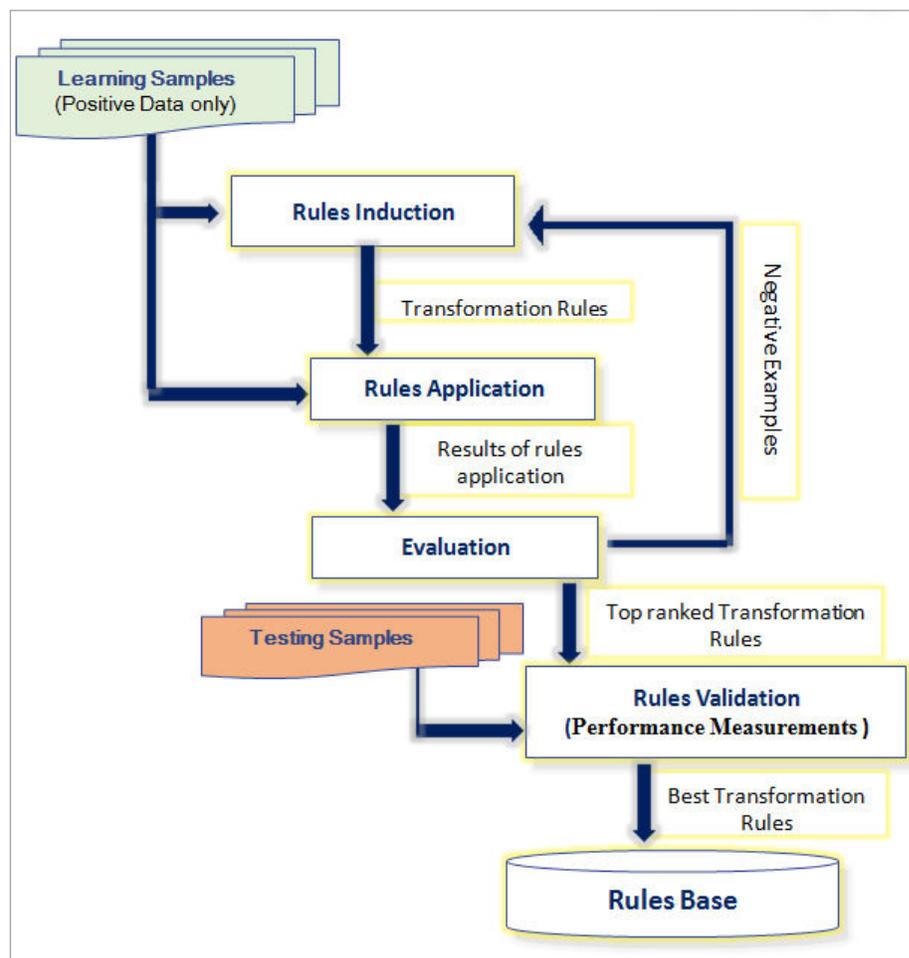


Figure 37: Learning from positive and negative data

In the context of MDD transformations, we learn using the negative examples in a different way. After inducing the transformation rules using the positive data only, we applied the rules on the training systems. Figure 37 presents the process followed by MTILP to consider the negative and positive examples. In the absence of the negative examples, MTILP starts by the learning the rules using the positive data only. Then the induced rules are applied on the learning samples (source models) to get the target models. In the generated target models the new artifacts that were not presented in the actual target models are considered as negative examples. Then the learning process is repeated again using the negative and positive examples.

7.6 Comparison with Prominent ILP Systems

This section compares MTILP system with some of the prominent ILP systems described in Section 2.3.3. The comparison criteria include: basic mechanism, search direction, incremental or batch system, single or multiple predicates learning, and the use of modes declarations and negative examples. The comparison between the proposed methods and the other well-known ILP systems is presented in Table 17.

Two search strategies can be defined in the context of ILP learner: either top-down or bottom-up. With the bottom-up strategy, the search process starts by using a specialized clause. Then it attempts to cover more positive examples by generalizing the clause till no more improvement is possible. On the other hand, with the top-down strategy, the search starts the search by using a general clause. Then it searches attempting to keep the clause more specific such that it avoids covering negatives.

In MTILP, the first step is constructing bottom clauses for all the positive examples. That is, it generates the most specific clauses to constrain the search space. Instead of searching for the atoms related to the examples in each iteration in the background knowledge, it searches in the specific clauses. One of the specific clauses is selected randomly and it starts generalization in a top-down manner. At each iteration, it adds one atom from the selected clause and checks the confidence value based on the other clauses.

Various approaches have been employed in the aforementioned systems, although most of them employ a covering approach. The idea is to start by selecting an example to be generalized. Then it searches to find a consistent clause covering the selected example. It generalizes the created clauses using the positive examples in such a way that is consistent when taking into account the negative examples. It removes all the covered positive examples. The generalization process is repeated for each selected example until all examples are covered. The presented systems employ different basic techniques. PROGOL and AELPH use inverse entailment method to implement the coverage approach. While FOIL employs a refinement graph for this purpose, GOLEM uses generalization Relative Least General Generalization (RLGG) method. Indeed, the search direction is a significant factor to control the selection of induction method. The top-down systems employ refinement graph and inverse entailment. They can start the induction using the most general clause. On the other hand, GOLEM starts the induction by constructing the most specific clauses.

In our approach, we cover the positive examples using different combinations. We aim to induce more rules that can be induced for some examples if they are combined before

deletions. In addition, to minimize the search space, the approach starts by constructing the most specific clause for a random selected example. In this way we, the approach starts in a bottom up manner, then it goes to generalize the clauses using the current combination of the positive examples in a top-down manner.

Considering the declarations of input/output modes is a useful feature for defining, in advance, structural biases in the induction systems. As explained in Chapter 6 when using ALEPH, it is mandatory to predefine the modes of the given predicate arguments. In addition, GOLEM, FOIL, and PROGOL use mode declarations in the same manner. In spite of the improvements gained due to the efficient pruning of the search space, occasionally it is not straightforward to correctly identify the model in advance. In some situations, it might be necessary to do trails with different alternatives and combinations.

Table 17: A comparisons of ILP systems

System	Basic Mechanism	Search Direction	Incremental	SPL	Interactive	Modes Declarations	Negative Examples
FOIL	Refinement Graph	Top-Down	No	SPL	No	Yes	Yes
PROGOL	Inverse Entailment	Top-Down	No	MPL	No	Yes	Yes
ALEPH	Inverse Entailment	Top-Down	No	SPL	Yes	Yes	Optional
GOLEM	RLGG	Bottom-Up	No	SPL	No	Yes	Yes
CIGOL	Inverse Resolution	Bottom-Up	No	SPL	No	Yes	Yes
MTILP	Genetic Algorithm	Top-down	Yes	MPL	No	Optional	Optional

In contrast, MTILP system does not require mandatory declarations mode. It allows learning based on the facts presented in the background knowledge and the given examples. The system gives the user the opportunity to predefine the modes if available. If not, the system is able to extract the possible head and body predicates from the given data.

The presence of negative data is considered one of the effective mean to prune the search space. Most of the induction systems require the use of negative data beside the positive ones to induce the needed hypotheses. However, one critical issue of using negative examples is that it may not be available in advance as we have seen in MDD context. GOLEM, FOIL, and PROGOL require the presences of negative data for the induction. Some ILP systems use close world assumption to generate the negative examples if they are not provided. Example of such systems is FOIL. On the other hand, ALEPH and the proposed system can run directly on data that have only positive examples. Definitely, the availability of negative data may help narrowing the search space.

CHAPTER 8

EMPIRICAL EVALUATION

As stated in Chapter 1, in this research we aim to utilize inductive learning to induce transformation rules through utilizing a set of examples where each example consists of source-target pair. Chapter 5 presented a model transformation paradigm that can employ an ILP system to induce, evaluate, validate and apply the transformation rules. In Chapter 6, ALEPH system has been used as rule learner. Due to many limitations showed by ALPEH in the context of MDD transformations, we propose a new ILP system, MTILP, described in Chapter 7 to overcome the investigated limitations.

In this chapter, we present the experimental work conducted using the MTILP system. The objective is to make an assessment of the MTILP system and to generate transformation rules to be used later for transformation of new source models. Different sets for learning and validation have been used to induce, evaluate and validate the induced theory. We present the results of reasoning, in terms of specificity, sensitivity, precision, f-measures and accuracy, when the rules generated by the MTILP system are applied.

We used the dataset and the transformation tasks used with ALEPH experiments. Then we compare the findings for the two different systems, namely ALEPH (presented in Chapter 6) and MTILP, in the context of MDD transformations and some other tasks. Thus the experiments were conducted based on the following objectives:

Obj1: Comparing the performance of MTILP against performance of ALEPH system in the context of MDD transformation rules induction.

Obj2: Evaluating MTILP System capability to induce different types of MDD transformation rules specially many-to-many rules.

Obj3: Evaluating the MTILP system performance in the context of other datasets.

8.1 Experiments and Results

A set of experiments have been conducted according to the objectives stated above. The datasets, transformation tasks and evaluation measures presented in Chapter 6 have been used in the following experiments. Moreover, we use the same learning and validation sets that were selected randomly. We aim here to evaluate the performance of the MTILP system and compare the obtained results with the results presented previously when using ALEPH system.

8.1.1 Packaging Transformation Rules

This section describes the experiments performed with inducing and applying transformation rules related to packaging the class diagrams. The goal of these experiments is to provide an assessment of the single predicate learning. The same sets of learning and validations sets presented in Chapter 6 are used in this experiment. The available examples have been used for learning and validation in the ratio 2:1, respectively. Although the sets were sampled randomly we intentionally utilize the same

sets, sampled in Section 6.3 , to be able to compare the induced rules and their performance.

8.1.1.1 Rules Induction Using Learning Examples

In this experiment we utilize 22 systems as input to MTILP system to generalize transformation rules. Each system comprises a pair of source-target models. MTILP does not require modes, types, determinations declarations. Thus, it is possible to feed all the learning examples with different size of packages. That means the given positive examples can have different arity.

Table 18: Samples of the induced transformation rules - Packaging transformation task

Induced Transformation Rule
$packageOfClasses(A, B, C) \leftarrow$ $inheritance(A, B), inheritance(A, C).$
$packageOfClasses(A, B, C, D) \leftarrow$ $association(D, B), association(D, C),$ $association(C, A).$
$packageOfClasses(A, B, C, D) \leftarrow$ $association(D, B), association(C, D),$ $association(B, A).$
$packageOfClasses(A, B, C, D) \leftarrow$ $association(D, C), association(C, B),$ $association(C, A).$
$packageOfClasses(A, B, C, D) \leftarrow$ $inheritance(A, C), inheritance(A, D),$ $inheritance(A, B).$
$packageOfClasses(A, B, C, D, E) \leftarrow$ $inheritance(D, A), inheritance(D, B),$ $inheritance(D, E), inheritance(D, C).$

The proposed model transformation system receives the examples of systems pairs then it generates the input requirements for MTILP automatically. After grouping them and checking the predicates dependency, each group is provided to induce the corresponding rules. Table 18 shows samples of the rules induced using the leaning systems. The same procedure presented in Section 5.5 is used here. After inducing the rules, we evaluate each rule individually against the all learning systems, batched together, to measure the rules' performance.

Using 22 learning samples, MTILP system was able to induce 63 hypotheses (rules). The accuracy measures of the rules are presented in Figure 38. Although we applied the rules on the samples used for learning, some rules show very low accuracy. Two reasons behind that. First, some rules came from a small set of positive examples and they are most-specific to those cases. When applying them on large data containing their cases, they are applicable to those cases only. However, it is not the case for all rules induced from small set of positive examples. Second, some rules application result in negative examples i.e. artifacts that do not exist in the actual source models.

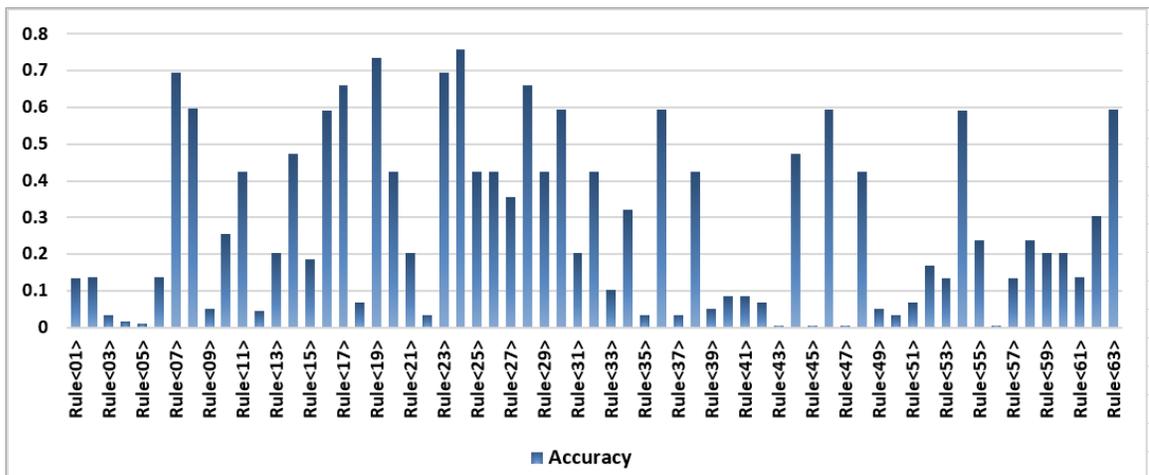


Figure 38: Accuracy measures of the induced rules (all the learning systems)

8.1.1.2 Evaluation of the Induced Rules

By rules evaluation we aim to select the best subset of the induced rules to be utilized by the practitioners to transform new given source instances. In the first step, we consider the resulting negative example to improve the learning process. As a result, some rules produce negative examples disappeared from the new rules. Second step concerns the ranking of all the resulting rules based on their accuracy measures as shown in Figure 39.

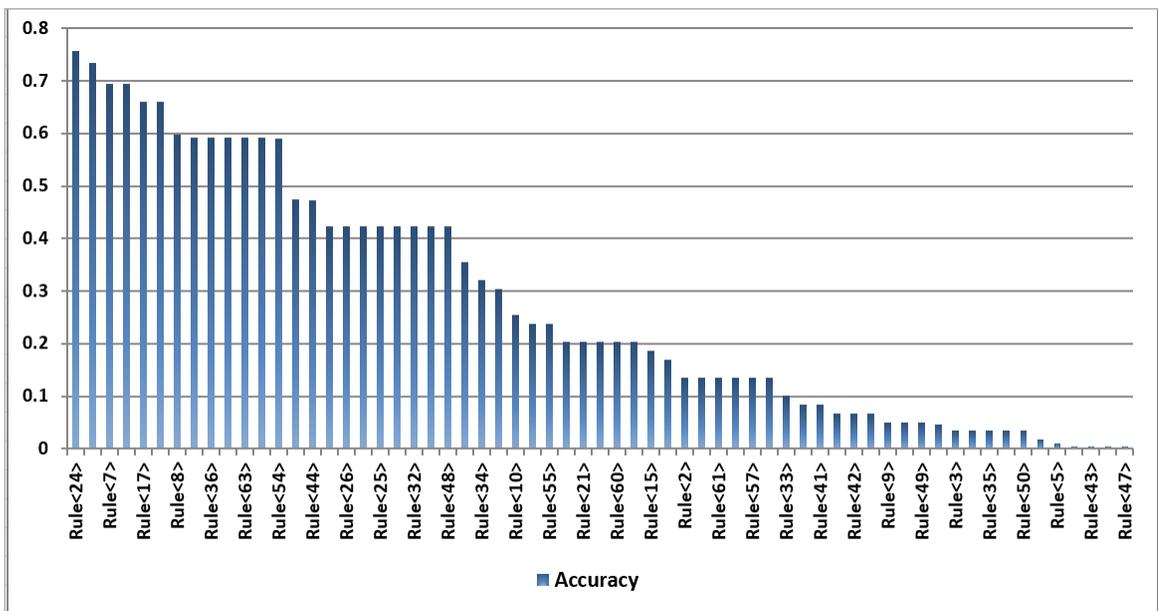


Figure 39: Ranking induced rules based on their Accuracy measures

Based on a specific accuracy threshold of 0.4 we select a subset consisting of 24 rules that provided the highest accuracy. This subset to be evaluated against the learning systems. In this step, the candidate 24 rules were applied on the learning systems one by one. To specify the best combinations of rules that can give best accuracy for each system, GA-based procedure has been used (Section 5.5.3.2). The resulting accuracy measures for each learning system are shown in Figure 40. The shown accuracies present the best combinations of the rules can be applied on the corresponding system.

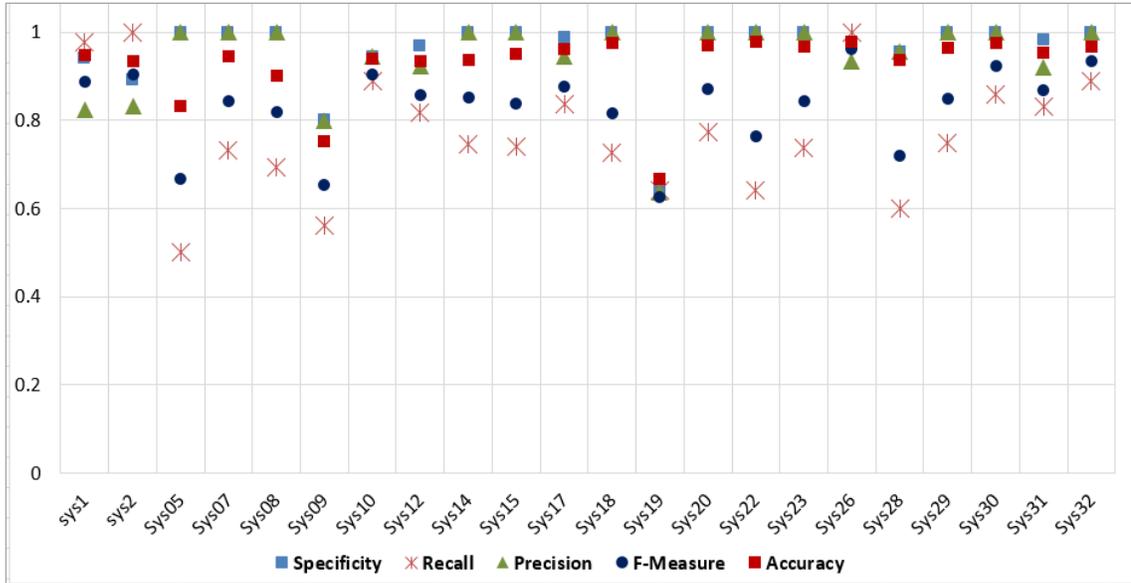


Figure 40: Evaluating the Induced Rules Using Learning Systems

Then we counted the number of times each rule was applied. Indeed, some rules are applied frequently with many systems when looking at the best results. This means they seem to be used as generic rules that apply to different systems. Thus the aim here is to count the frequency of rules applications and find the percentage by dividing the application times by the number of systems. Figure 41 shows the application time for the used 24 rules.

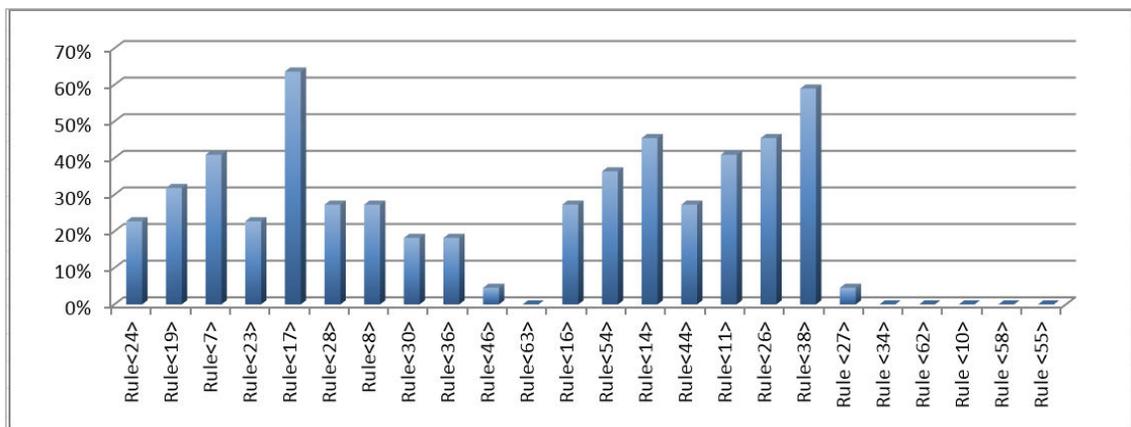


Figure 41: Percentage of the application frequency that rules selected to give best result

The percentage of rules application will be used as a guide when we want to apply the rules on new unseen samples or for future development. Figure 42 shows the rules ranked using the applications times. It is obvious that some rules have been selected many times. For instance, *Rule <17>* has been selected with 60% of the systems to give the best results. On the other hand, we have around 8 rules that were not selected for applications to get the good accuracy. Six rules have not been selected any time with learning examples. Thus in the next phase we excluded such rules that were not selected any time.

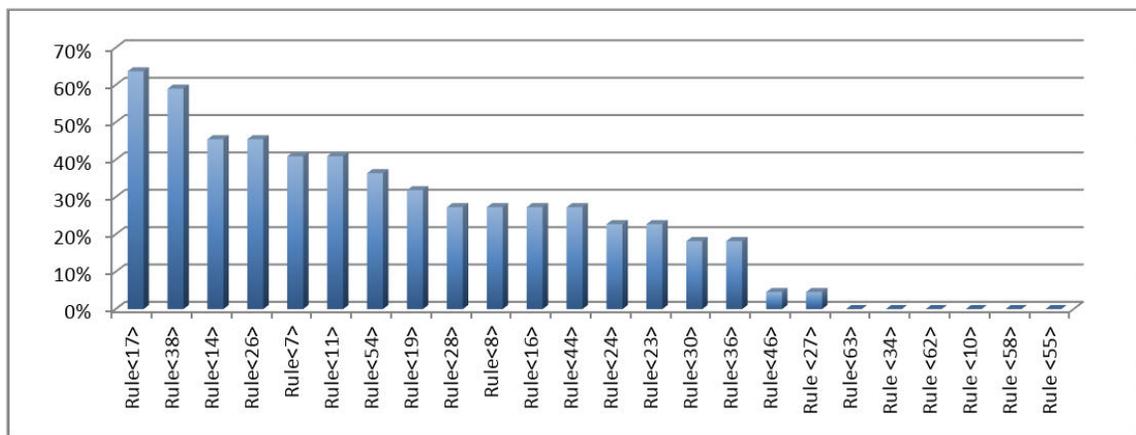


Figure 42: The set of 24 rules ranked based on their application frequency

8.1.1.3 Rules Validation

The ranked rules presented in Figure 42 were validated using the validation set that consists of 12 systems. Several validation experiments have been conducted. In each experiment a subset of the ranked rules were applied on the validation systems one by one. The first experiment used only two top ranking rules. In the second experiment, one more rule was added to the subset, and so on. Figure 43 shows the averages (of all validation systems) for the different experiments.

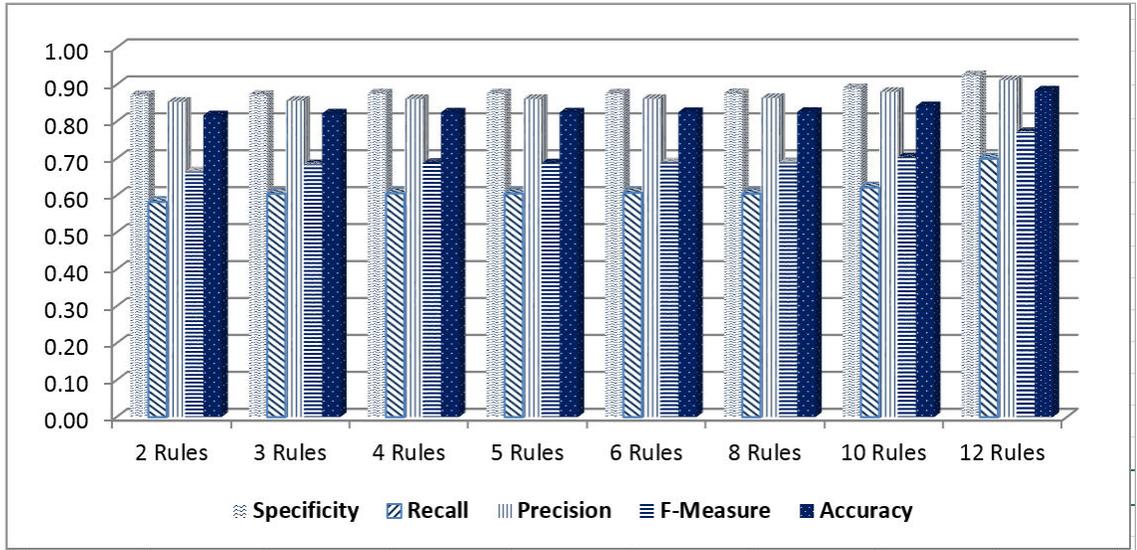


Figure 43: Selection of the rules for validation

As shown in the above figure, the application of different subsets of the rules give a comparable results. However, it is worth noting that the figure demonstrates the overall average of accuracy measures resulting from using 12 examples. In fact, increasing or decreasing the number of rules varies from one system to another. For example, when applying a subset contains 5 rules the results of some systems were improved while others were impaired. The resulting changes are very low, thus when calculating the average we get close values.

As additional validation of the induced rules, several experiments were conducted using random rules selection. Three different runs were considered, in each a number of 7 rules have been selected randomly. Figure 44 shows the averages (accuracy measures) for each experiment.

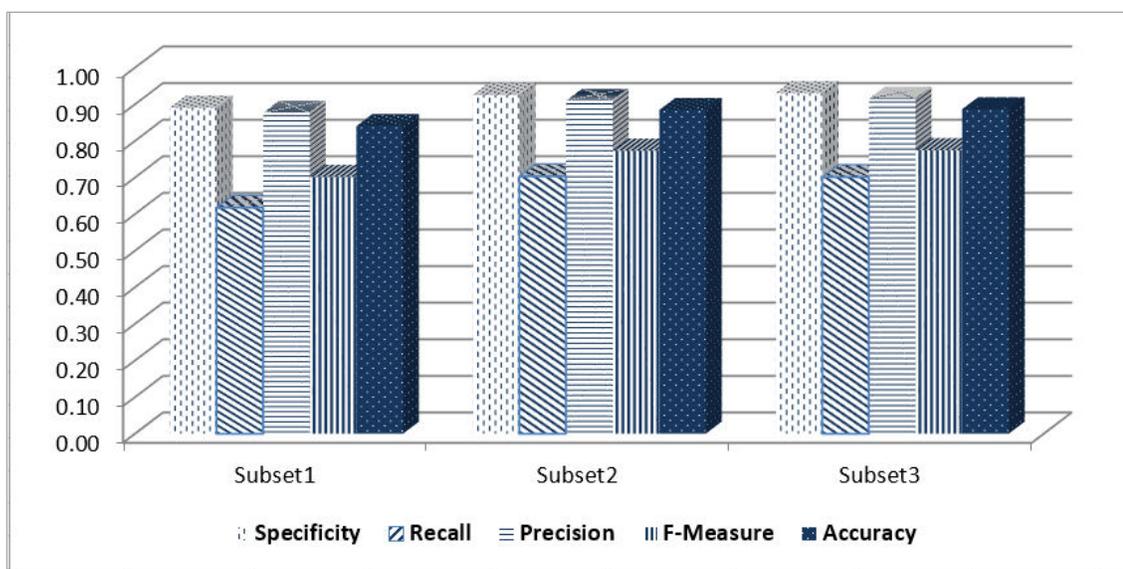


Figure 44: Selection of the rules randomly for validation

8.1.2 Introducing Façade Rules

As explained in Section 6.4.3, in the used datasets only 13 systems use façade design practices. The available examples have been used for learning and validation in the ratio 2:1, respectively. The same subsets and the same procedure followed when using ALEPH have been considered here too.

Indeed, introducing a façade means introducing more than one artifact, not only the façade interface. These new artifacts include: a new interface, associations linking the external clients to the interface, and associations linking the interface to the classes placed in its package. Figure 45 demonstrates a pair of source-target model that considers a façade design. The set of examples used to learn the rules have different forms of this practice. For instance, in the form presented in Figure 45, *Package 1* receives four external references, from a single package, to its classes. Another form may show a package that receives five external reference come from more than one packages.

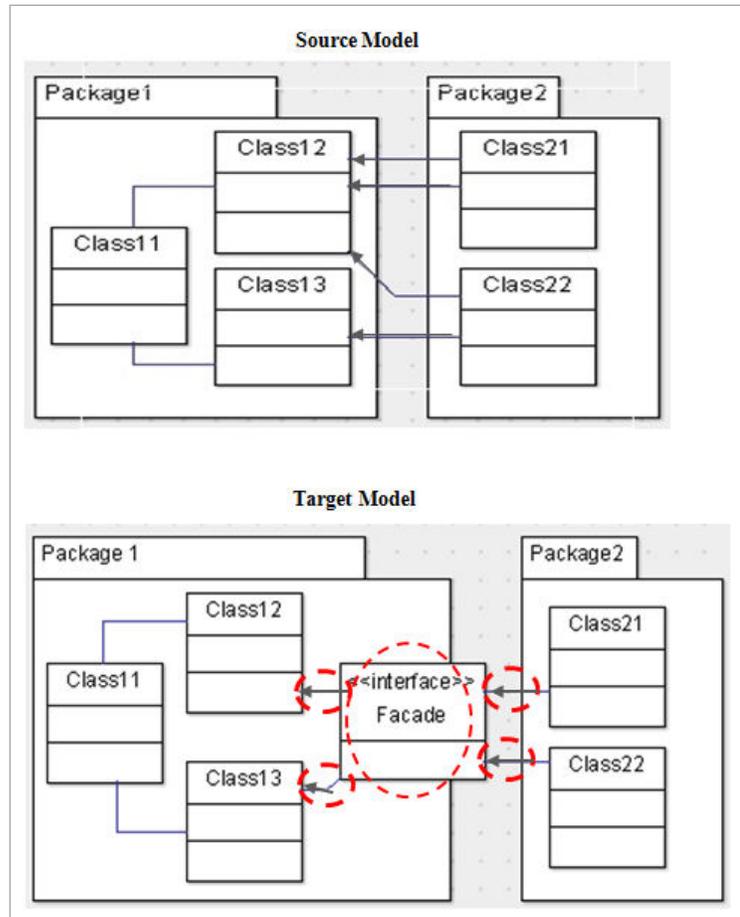


Figure 45: New artifacts when introducing façade interface

In Figure 45, obviously the two kind of relationships added to the model should be linked to the new interface (Façade). That means the new artifacts will be added in a particular order where the interface should be added first then the other relationships. In other words, there is a need to add the facts that describe the façade interface to the background knowledge to be able to induce the rules related to the associations. Updating the background knowledge with new facts is called a *theory revision*, described in Section 0.

In Chapter 6 we considered this case study as a type of single predicate learning due to the limitation of ALEPH system in this regard. When using ALEPH system, we noticed

that the rules provided were a kind of a suggestion to introduce a façade for any package having an external relation. To learn the rules related to adding the appropriate relations, it is needed to run the engine with the needed inputs (background knowledge, modes and types declarations, and examples) regardless of the other related artifacts. This is because, ALEPH does not provide incremental learning in the way the dependency can be discovered and the theory can be updated. The user has to update the background knowledge after each run.

8.1.2.1 Rules Induction: Incremental Learning

When introducing a façade more than one artifact can be added to the corresponding model. These resultant artifacts cannot be added in any order. That means there is a particular order to add these artifacts. For instance, the façade is added before the different relationships that linking the classes to the interface. In the learning context, different sets of positive examples, with different predicates, are extracted from the learning examples as shown in Figure 46.

```
packageHasFacade(packageA, interfaceA).  
associationFromClassToFacade(packageB, classC, packageA, interfaceA).  
associationFromFacadeToClass(packageA, interfaceA, packageA, classD).
```

Figure 46: The positive examples related to introducing a façade

The learning process starts with the given background facts and the first group positives based on initial order (as explained in Chapter 7). It may try different orders till it finds the most appropriate one.

Table 19: Samples of the induced transformation rules - Introducing Façade

Design Activity	Induced Transformation Rule
Façade artifact	$packageHasFacade(A, B) \leftarrow$ $package(A), packageHasClass(A, C), packageHasClass(A, D),$ $interfaceFacade(B), association(E, F, A, C), association(E,$ $G, A, D), packageHasClass(E, F), packageHasClass(E, G).$
	$packageHasFacade(A, B) \leftarrow$ $package(A), packageHasClass(A, C), packageHasClass(A, D),$ $interfaceFacade(B), association(E, F, A, C), association(E,$ $G, A, C), packageHasClass(E, F), packageHasClass(E, G),$ $association(H, I, A, D), packageHasClass(H, I).$
	$packageHasFacade(A, B) \leftarrow$ $package(A), packageHasClass(A, C), packageHasClass(A, D),$ $packageHasClass(A, E), interfaceFacade(B), association(F,$ $G, A, D), association(F, H, A, E), packageHasClass(F, G),$ $packageHasClass(F, H), association(I, J, A, E),$ $association(I, K, A, F), packageHasClass(I, J),$ $packageHasClass(I, K).$
Association between a client and façade.	$associationFromClasstoFacade(A, B, C, D) \leftarrow$ $packageHasClass(A, B), packageHasFacade(C, D),$ $association(A, B, C, E), packageHasClass(C, E).$
Association between a façade and a class	$associationFromfacadetoClass(A, B, A, C) \leftarrow$ $packageHasClass(A, C), packageHasFacade(A, B),$ $packageHasClass(D, E), association(D, E, A, C),$ $associationFromClasstoFacade(D, E, A, B).$

Table 19 presents a sample of rules induced for introducing a façade, and its related relationship namely *associationFromClasstoFacade* and *associationFromfacadetoClass*. The first group shows sample of the rules induced for introducing façade, while the other two groups shows the rules for the associations artifacts. It is worth mentioning that, only one rule was induced for each association artifact. The last two rules are supposed to be

applied whenever there is application of any of the façade rules, i.e. their accuracy is related to the accuracy of applying the façade rules. The induced rules were applied (one by one) to the learning systems (source models) and the results were compared against the actual target models. Figure 47 demonstrates the accuracy measures of each façade design rule. After applying any façade rules the two other rules (related to the consequences relations) are applied. Thus the accuracy measures show a combination of the results of the three rules together. Indeed, if there is no façade rule triggered, none of the association rules is applied. That is, the application of the rules related to the associations is a consequence of firing any rule of introducing façade.

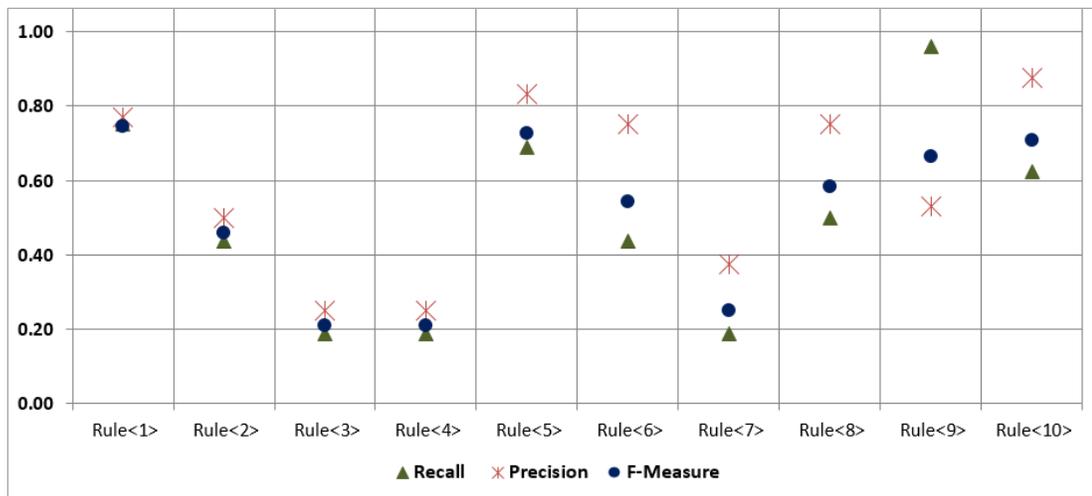


Figure 47: Accuracy measures for the induced rules

8.1.2.2 Rules Evaluation

The rules evaluation focuses only on the first group of the transformation rules. That is the rules of introducing façade ($packageHasFacade(A, B) \leftarrow \dots$) have been considered in this procedure. As explained above the other two groups application rely on applying the first group of rules. If there is a new façade introduced to the package, the needed

relations are added correctly. Thus, the accuracy shown in Figure 48 of the rules depends on the first group only. After the application of the rules on the learning examples, some rules are applied perfectly many times, while others (such as *Rule<3>* and *Rule<4>*) might be used one time. Next step in the evaluation procedure, all the rules are ranked based on their application frequency. The predefined threshold we considered to take the most frequent applied rules was (0.3).

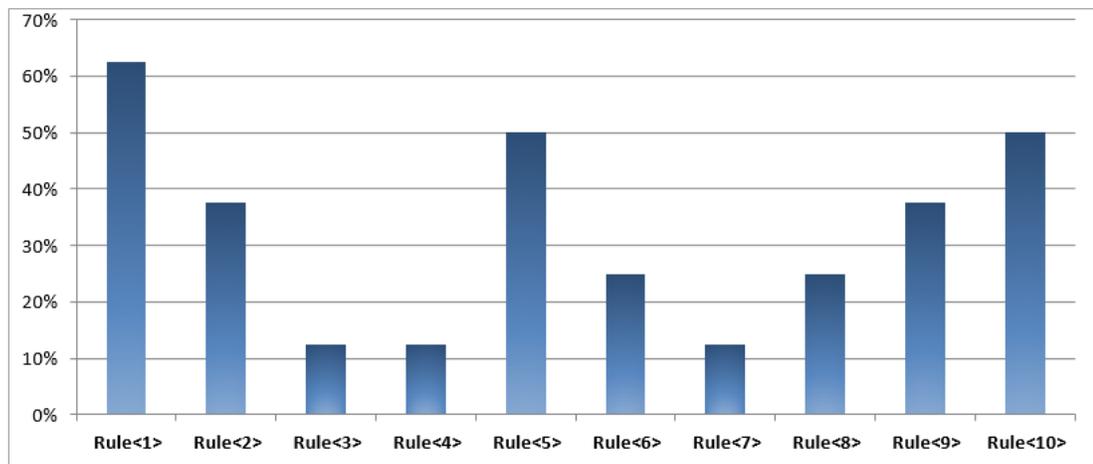


Figure 48: Frequency of the rules application – Façade Transformation

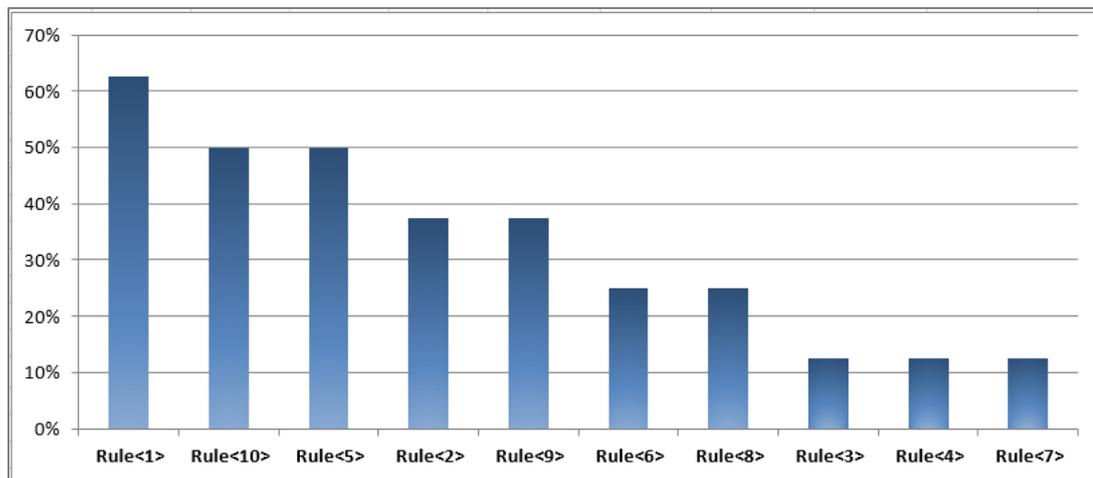


Figure 49: Introducing Façade rules ranked based on their application times

As a result only the top five rules were selected. When looking for the best accuracy, these three rules have been selected to be applied on more than 30% of the presented systems as shown in Figure 49.

8.1.2.3 Rules Validation

This section shows the application of the top ranked rules (shown in Figure 49) on the validation set. Figure 50 demonstrates the obtained results from the application of different rules. Again the results show also the application of the rules related to *accociationFromClasstoFacade* and *accociationFromfacadetoClass*.

Figure 50 shows that using a subset that consists of 3 rules gives the same accuracy as using subsets have 4 and 5 rules, which means that we can utilize only the smallest set without more application overhead. In other words, the top three rules were selected $\geq 50\%$ times to give best results. Other experiments were conducted by selecting randomly two and three rules. It is obvious from Figure 50 that best accuracies obtained when applying the top-two and the top-three rules.

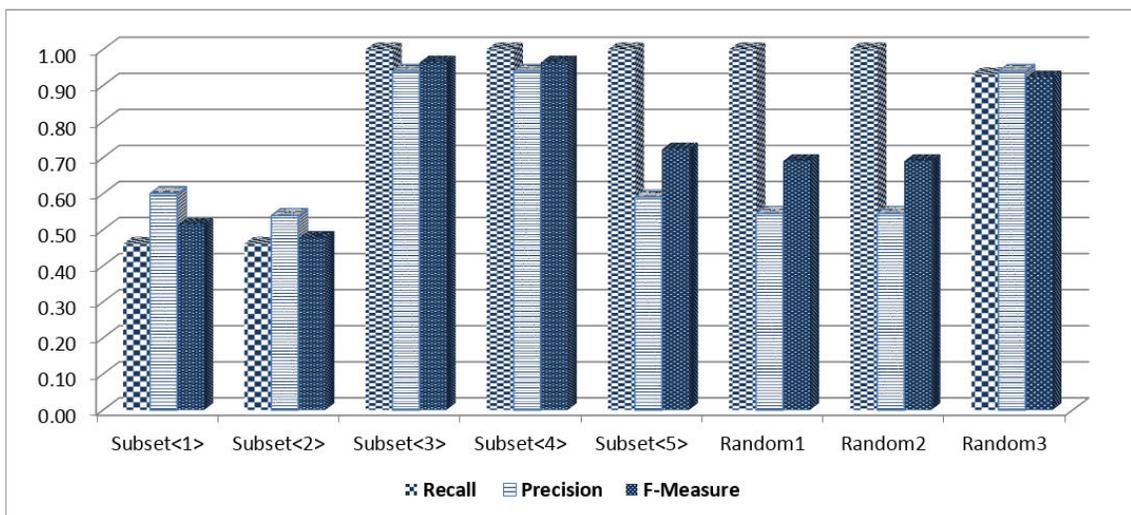


Figure 50: Validation results – various combinations

8.2 More Model Transformation Tasks

In this section, we investigate another two model transformation tasks: (1) detecting and eliminating a Blob anti-pattern; and (2) splitting a large class introduced in the analysis phase to more than one class in the design phase. In the following we introduce more descriptions and present the rules induced using hypothetical examples.

8.2.1 The Blob Anti-pattern

The Blob is a well-known anti-pattern that usually needs to be refactored. This anti-pattern is characterized by a class diagram composed of a single controller class that is surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class. The Blob anti-pattern can be found in the design. Thus, it is expected to see the source models as designs that have Blob anti-pattern (as shown in Figure 51), while the target model is shown after performing the required refactoring (as demonstrated in Figure 52).

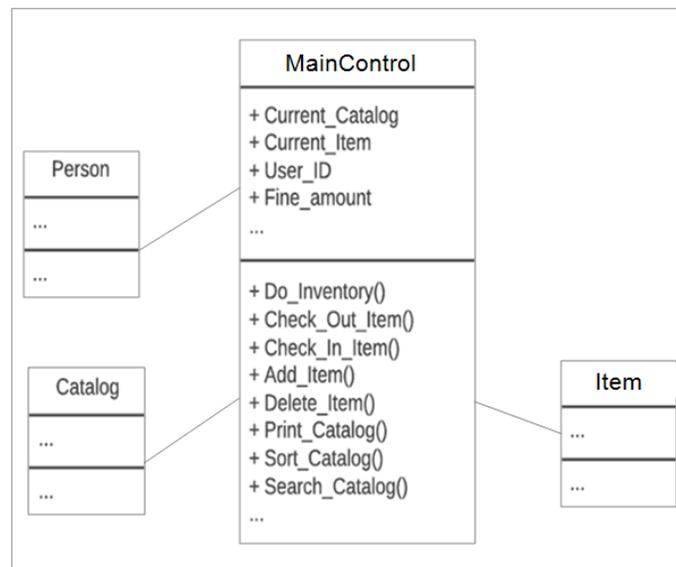


Figure 51: The Blob anti-pattern (Source model)

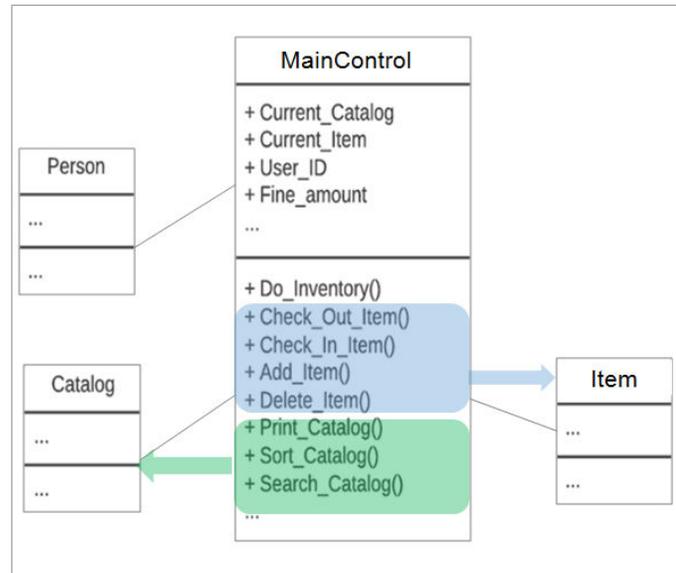


Figure 52: The Blob anti-pattern (Target model)

In this example in the class called “*MainControl*” there are a number of methods (colored in blue) that use extensively data (attributes) presented in class “*Item*”. On the other hand, there are a number of methods (colored in green) that use data presented in class “*Catalog*”. Thus, the two groups of methods are moved to the destination classes based on the data usage.

Since the given XMI files are converted into logic program, each presented artifact is any given model can be represented by a first-order predicate. In this experiment, beside the class diagram we utilized the sequence diagram as input as shown in Figure 54. The sequence diagram is used as input in order to provide more information about which attributes are used by which methods.

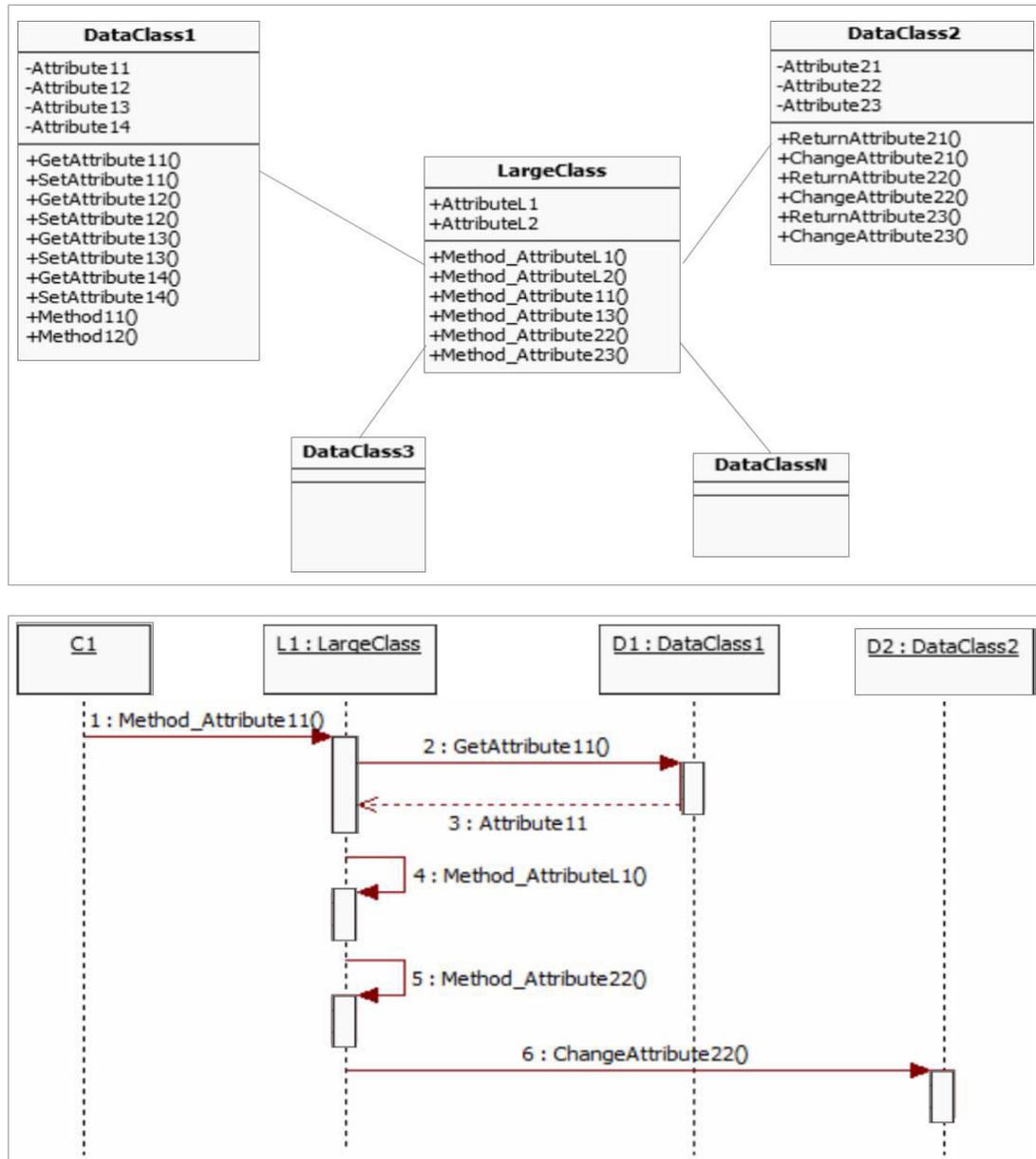


Figure 53: Snapshot of the class and sequence diagram used as learning example

MTILP was able to induce a suitable rule for the anti-pattern based on the provided examples. For this and the next experiment we just investigate the ability of MTILP to solve various transformations tasks. In other words, the proposed model transformation phases are not employed here. The same input has been provided to ALEPH system.

```

class(largeClass). classHasAttribute (largeClass,attributeL1). classHasAttribute
(largeClass, attributeL2). classHasMethod (largeClass, method_attributeL1).
classHasMethod (largeClass, method_attributeL2). classHasMethod (largeClass,
method_attribute11). classHasMethod ( largeClass method_attribute13).
classHasMethod ( largeClass, method_attribute22). classHasMethod ( largeClass,
method_attribute23). class(dataClass1). classHasAttribute (dataClass1,
attribute11). classHasAttribute (dataClass1, attribute12). classHasAttribute
(dataClass1, attribute13). classHasAttribute (dataClass1, attribute14).

.....
methodUsesAttribure (largeClass, method_attribute11, dataClass1,attribute11).
methodUsesAttribure (largeClass, method_attribute22, dataClass1,attribute22).
association(largeClass, dataClass1). association(largeClass, dataClass2).
.....

```

Figure 54: The Blob anti-pattern - Problem Representation

Table 20 shows the rules induced by using ALEPH and MTILP systems. It is obvious from the induced rules, ALEPH considers only one relation in the body of the rule. That is whenever, there is a method that uses an attribute in another class move the method to that class. However, in the given examples at least two relations are considered to move the method to another class as shown by the rule induced by MTILP.

Table 20: Rules induced using ALEPH and MTILP systems

System	Induced Rule
ALEPH	$classOwnsMethod(A,B) \leftarrow methodUsesAttribure(C,B,A,D).$
MTILP	$classOwnsMethod(A,B) \leftarrow class(C), classHasMethod(C,B), class(A), classHasAttribute(A,D), methodUsesAttribure(C,B,A,D), classHasAttribute(A,E), methodUsesAttribure(C,B,A,E).$

8.2.2 Splitting a Class into Partial Classes

In the analysis phase one or more large class/classes are introduced where the large class has a large number of methods. Such class is subject to be divided into more than one class in the design phase. Figure 55 shows a hypothetical example for splitting a large class. The methods interacting extensively with each other are grouped together in a new class. Figure 56 shows a snapshot of the background knowledge extracted from hypothetical example. Sequence diagram can be to provide more information about the methods interactions.

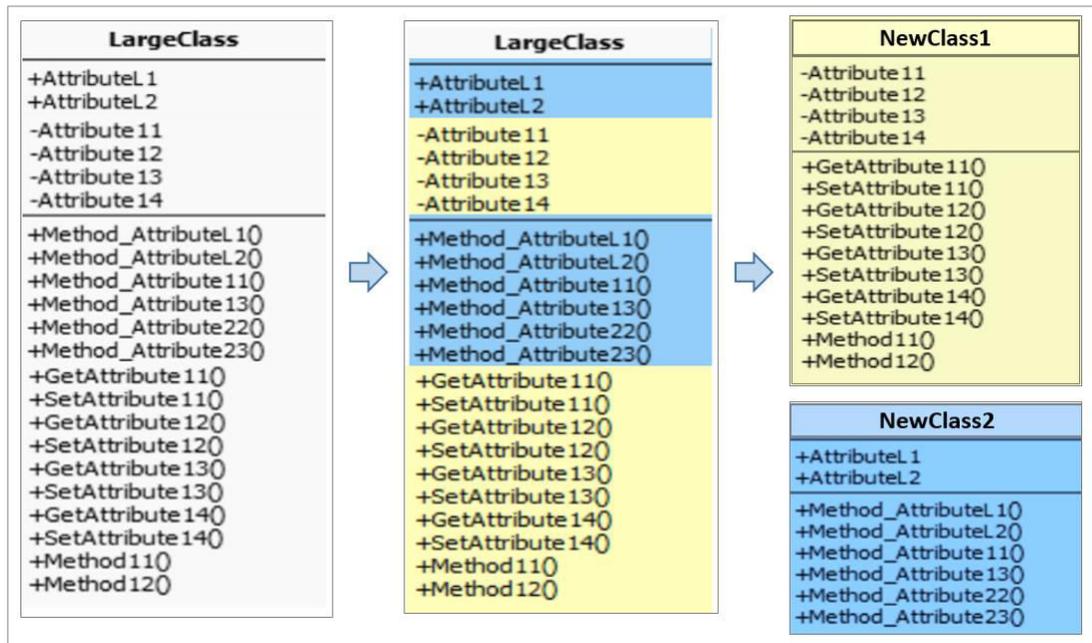


Figure 55: A Example of splitting class– source and target models

```

class(largeClass). classHasAttribute (largeClass,attributeL1). classHasAttribute
(largeClass, attributeL2). classHasMethod (largeClass, method_attributeL1).
classHasMethod (largeClass, method_attributeL2). classHasMethod
(largeClass, method_attribute11). classHasMethod ( largeClass
method_attribute13). classHasMethod ( largeClass, method_attribute22).
classHasMethod ( largeClass, method_attribute23). classHasMethod
(largeClass, getattribute11). classHasMethod (largeClass, getattribute12).
classHasMethod (largeClass, getattribute13). classHasMethod (largeClass,
getattribute14).
classHasMethod (largeClass, setattribute11). classHasMethod (largeClass,
setattribute12). classHasMethod (largeClass, setattribute13). classHasMethod
(largeClass, setattribute14). classHasMethod (largeClass, getattribute11).
classHasMethod (largeClass, getattribute12). classHasMethod (largeClass,
method11). classHasMethod (largeClass, method12).
.....
methodCalls (largeClass,method11, largeClass, method12). methodCalls
(largeClass, method_attributeL1, largeClass, method_attributeL2).
methodCalls (largeClass, method_attribute11, largeClass,
method_attribute12). .....

```

Figure 56: A Snapshot of the background knowledge - Splitting class example

Two groups of positive examples are provided in this transformation problem. The first group has the predicate *newClass(className)* while the second has the predicate *newClassHasMethod(className, methodName)* as shown in Figure 57.

```

newClass(newClass1).
newClass(newClass2).
...
...

newClassHasMethod(newClass1, get attribute11).
newClassHasMethod(newClass1, set attribute11).
newClassHasMethod(newClass1, get attribute12).
newClassHasMethod(newClass1, set attribute12).
.....
.....
newClassHasMethod(newClass2, method_attributeL1).
newClassHasMethod(newClass2, method_attributeL2).
.....
.....

```

Figure 57: A snapshot of the presented positive examples

When providing the background facts and the positive examples to ILP learner in the presented order i.e. *newClass* predicate first, no solution can be induced. That is, it is required to learn the examples in the appropriate order which is provided by MTILP. When starting with a given order, for instance, *newClass* first then *newClassHasMethod* second. No rule is induced because no atoms returned when searching about "class1" or "class2" in the background facts. In such case, MTILP automatically generated another order and tried to solve the problem starting by another predicate *newClassHasMethod*. Sample of the induce rule is presented in Figure 58.

```

newclass(A) ←
  newClassHasMethod(A,B), newClassHasMethod(A,C).
  newClassHasMethod(A,D). classHasMethod(E,B),
  classHasMethod(E,C), classHasMethod(E,D),
  methodCalls(E,B,E,D), methodCalls(E,B,E,C),
  methodCalls(E,C,E,B), Class(E).

```

Figure 58: Sample of the induced rule -Splitting Class

8.3 Application of MTILP in Other Contexts

In this section, we test the capability of MTILP against other datasets i.e. different from the MDD transformations problem.

8.3.1 Learning Transitive Rules

In this experiment we test the capability of the proposed system in inducing transitive rules. Given the target predicate $t(A_1, A_2 \dots, A_n)$ and the background knowledge predicates $f_1(B_1, B_2 \dots, B_m), \dots, f_r(Z_1, Z_2 \dots, Z_k)$. To find the transitive rule: the target predicate $t(A_1, A_2 \dots, A_n)$ represents the head of the rule, while the body of the rules can

be synthesis using the background facts. To find the related facts, the arguments of the target predicate are used. However, in some cases it is required to add the indirect related facts to the rule body. For instance, for the background facts $\{r(y, z), q(v, l), t(z, w, v)\}$ and the target predicate $p(x, y)$, the following a transitive rule can be induced: $p(A, B) \leftarrow r(B, C), t(C, D, E)$, where A, B, C, E and E are variables. MTILP system provides a setting to adjust the relevance level. To determine if it is needed to include the direct related and indirect related facts. Further discussion about the related and unrelated facts can be found in Chapter 7.

Michalski's train problem is a typical problem in learning transitive rules. Figure 59 demonstrates a description of the train problem. Assume ten railway trains: five are travelling east and five are travelling west; each train comprises a locomotive pulling wagons; whether a particular train is travelling towards the east or towards the west is determined by some properties of that train.

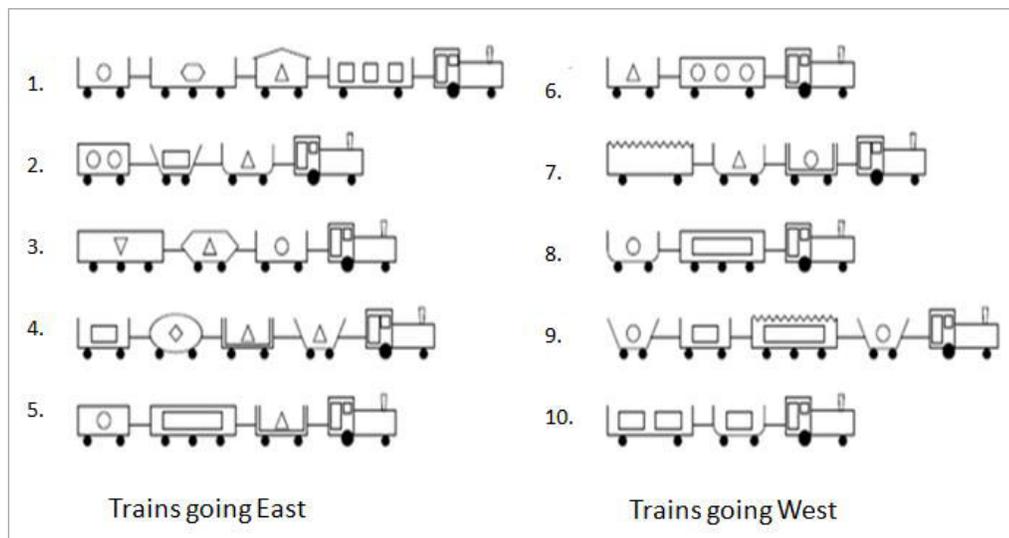


Figure 59: Michalski's original ten trains

The learning task: determine what governs which kinds of trains are Eastbound and which kinds are Westbound. It is a kind of classification problem to induce a rule that classify if the train goes towards west or towards east. Michalski's train problem can be viewed as a classification task: the aim is to generate a classifier (theory) which can classify unseen trains as either Eastbound or Westbound.

Table 21: The representation of Michalski's train problem

Input Type	Logic Program statements
Background Knowledge	<pre> % eastbound train 1 short(car_12). closed(car_12). long(car_11). long(car_13). short(car_14). open_car(car_11). open_car(car_13). open_car(car_14). shape(car_11,rectangle). shape(car_12,rectangle). shape(car_13,rectangle). shape(car_14,rectangle). load(car_11,rectangle,3). load(car_12,triangle,1). load(car_13,hexagon,1). load(car_14,circle,1). wheels(car_11,2). wheels(car_12,2). wheels(car_13,3). wheels(car_14,2). has_car(east1,car_11). has_car(east1,car_12). has_car(east1,car_13). has_car(east1,car_14). </pre>
Positive Examples	<pre> eastbound(east1). eastbound(east2). eastbound(east3). eastbound(east4). eastbound(east5). </pre>
Negative Examples	<pre> eastbound(west6). eastbound(west7). eastbound(west8). eastbound(west9). eastbound(west10). </pre>
Induced Hypothesis	<pre> eastbound(A) ← has_car(A,B), close(B), short(B). </pre>

As any ILP problem, to start learning it is required to provide the background facts describe the trains. The following knowledge about the cars belong to each train can be extracted: which train it is part of, its shape, how many wheels it has, whether it is open (i.e. has no roof) or closed, whether it is long or short, the shape of the things the car is loaded with. In addition, for each pair of connected wagons, knowledge of which one is in front of the other can be extracted.

Different predicates are used for description purpose such as if it the train has a car, and the characteristics of the cars (short, long, closed, open, etc.). In Table 21 we show the background knowledge for train east1 as an example. Cars are uniquely identified by constants of the form *car_x_y*, where x is number of the train to which the car belongs and y is the position of the car in that train. For example *car_12* refers to the second car behind the locomotive in the first train.

Michalski's train problem is a typical case for learning transitive rules. When learning the target relation *eastbound(train)* is only related with *has_car(train, car)* relation. The other background relations are only related with *has_car* relation.

In the problem representation, the *eastbound* relation has 5 records and the system takes the first target instance (*east1*). The target relation has one parameter and its type is train. Only (*has_car*) relation is related with *eastbound* and the other background relations are not related.

MTILP system supports the ability to adjust the setting of the relevance level to consider or ignore the unrelated background knowledge facts. When the relevance level is 0, MTILP induces very specific rules that do not reflect the characteristics of the train

concept. An instance of the induced rule is $eastbound(A) \leftarrow has_car(B, car11)$ which does not include any information about the properties of the *cars* of the train. Thus it is needed to adjust the relevance level to 1 to be able to consider the indirectly related facts in the induced rules.

8.3.2 Patrick Winston's arch problem

Patrick Winston implemented a program for learning the arch concept. The learning process considered various examples that show arch figures. The examples consist of both positive and negative examples of arches. Figure 60 demonstrates examples of the arch problem.

Similar to inductive learning, the program starts from a particular example, then it generalizes through considering the presented examples and so on.

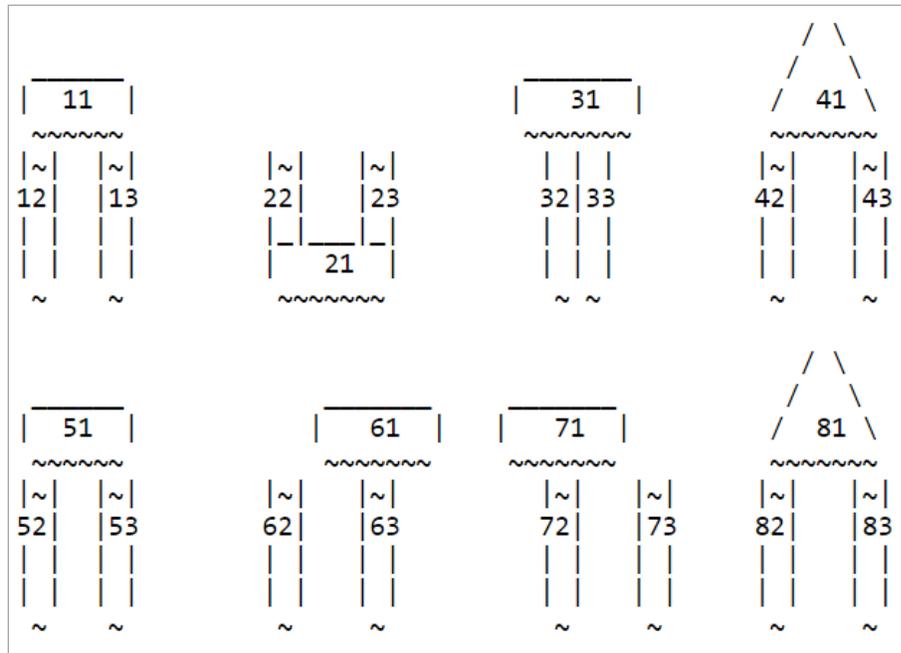


Figure 60: Example of Patrick Winston's arch problem

Although this problem appears simple when compared with other problems presented above, we select this problem for the sake of adaptation. That is, we add more information and relations to the examples to make the problem more complex. Then we present the problem to ALEPH and MTILP systems to compare their performances in such problem.

Table 22 demonstrates the results of using ALEPH and MTILP systems in rules induction for this problem. Although both systems were able to solve this problem, MTILP provided more expressive hypothesis.

Table 22: Induced rules - Arch Problem

System	Induced Rule
ALEPH	$arch(A,B,C) \leftarrow supports(B,A), supports(C,A).$
MTILP	$arch(A,B,C) \leftarrow brick(A), brick(B), left_of(B,C), brick(C), supports(B,A), supports(C,A).$

8.4 MTILP vs. ALEPH

This section compares the performance of MTILP against ALEPH system according to the experiments conducted in Chapter 6 and Chapter 8. Part of the experiments focused on inducing the transformation rules for two important transformation tasks: packaging the class diagram and introducing façades.

Various performance measures have been used in through this work to evaluate the accuracy of the induced rules. Five different measures have been used to assess the performance of the packaging rules. On the other hand, only precision, recall and F-measure have been used in the case of introducing Façade interface to the design. This is because the inability to calculate the negative true values. For the additional transformation tasks, presented in Chapter 8, due to the lack of data we would be content with the ability of the rules induction when comparing ALEPH and MTILP.

Thus, here, we present a comparison that shows the accuracy measurements of learning and validation results in both transformation tasks.

For the packaging transformation task, ALEPH was able to produce rules for packaging the class diagrams based on the given examples. We discussed the limitations encountered during the induction using ALEPH system. In addition to the preparation of the files and models declarations in each run, for packages with many classes ALEPH was not capable to induce rules. For packages with many classes and complicated relations ALEPH could not find a solution using different settings of search.

Figure 61 demonstrates the performance results for the packaging transformation task. It is obvious from the figure the highest accuracy values in both phases (learning and validation) obtained through MTILP. In the learning phase, the accuracy measure is 0.93 comparing with 0.89 using ALPEH system. Similarly, for the validation systems, the accuracy of MTILP is 0.88, while the accuracy of ALPEH is 0.83.

Definitely, the number of induced rules varies from one system to another. MTILP induced 63 rules comparing with 25 rules induced using ALEPH. The increase of the

induced rules came from the coverage of all the positive examples using different combinations. We explained that, not all the induced rules can be stored in the rule base for future development. The rules were evaluated in different ways and validated to select the best subset to be stored in the rule base. That mean only 12 out of 63 rules are recommended to be stored in the rule base.

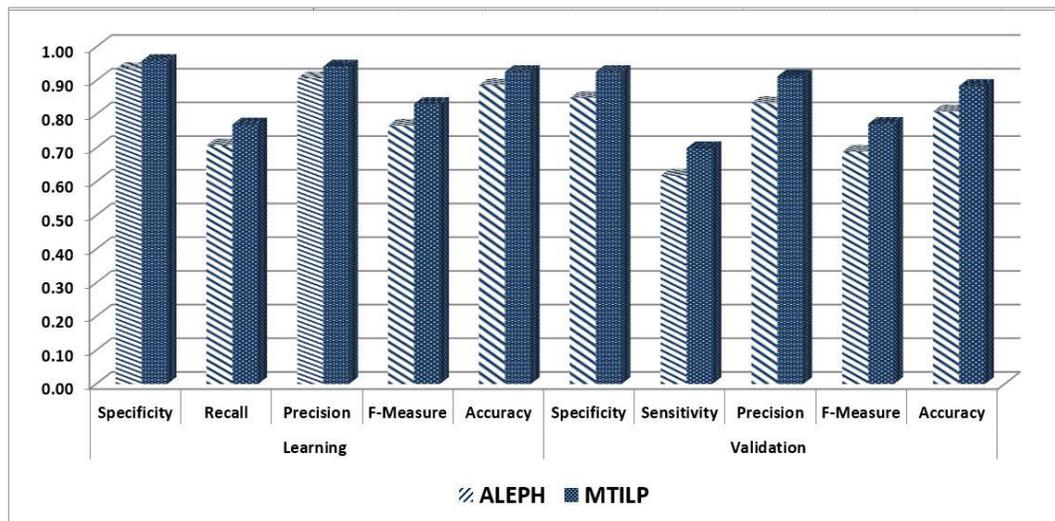


Figure 61: Introducing packages to the class diagram

Figure 62 demonstrates the measurements values in the context of the task of introducing façade interface. It is obvious from the figure the highest accuracy values in both phases (learning and validation) obtained through MTILP. In the learning phase, the f-measure is 0.98 comparing with 0.70 using ALPEH system. Similarly, for the validation systems, the f-measure obtained using MTILP is 0.96, while the f-measure of ALPEH is 0.66. It is important to mention that, the results obtained from MTILP came from a combination of applying façade and associations rules. However, the comparison with rules induced by ALEPH for façade interface is reasonable. The reason is that the application of the association rules is a consequence of the façade rules application. In other words, the

associations' artifacts will be added correctly to the model if the façade interface was added to the right package. Similarly, if there is a missed façade interface in the model, consequently all the related associations will not be added.

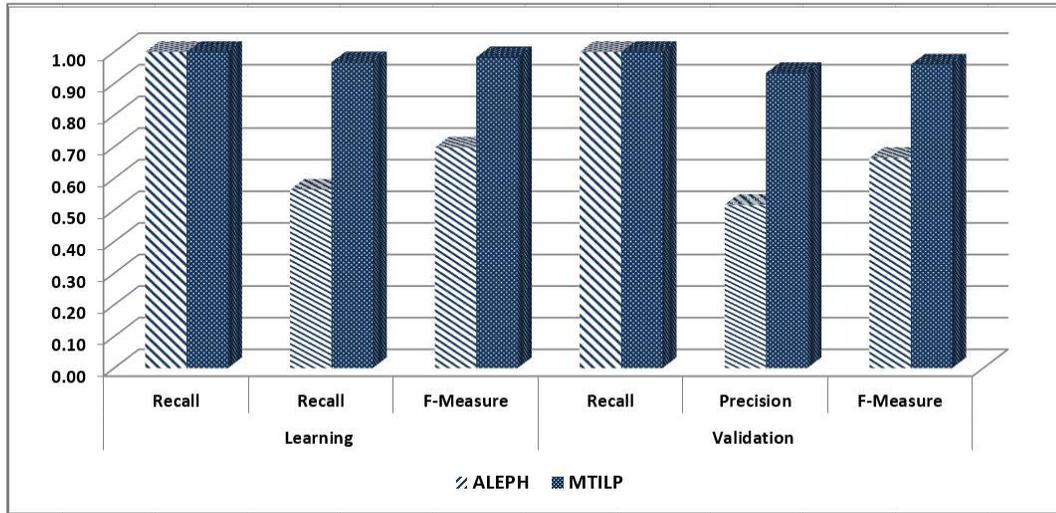


Figure 62: Introducing Façade design pattern

From the previous experiments, we notice that MTILP and ALEPH have common characteristics where there are many problems that can be solved correctly using both systems such as classification, learning transitive rules, arch problem without modifications, and packaging of class diagrams. However, ALPEH could not find solutions for some problems such as introducing façade interface, and modified version of arch problem.

As shown in Figure 63 , MTILP is not a replacement of ALEPH where ALPEH has various functionalities to deal with different problems related to decision tree learning, abduction learning and many others.

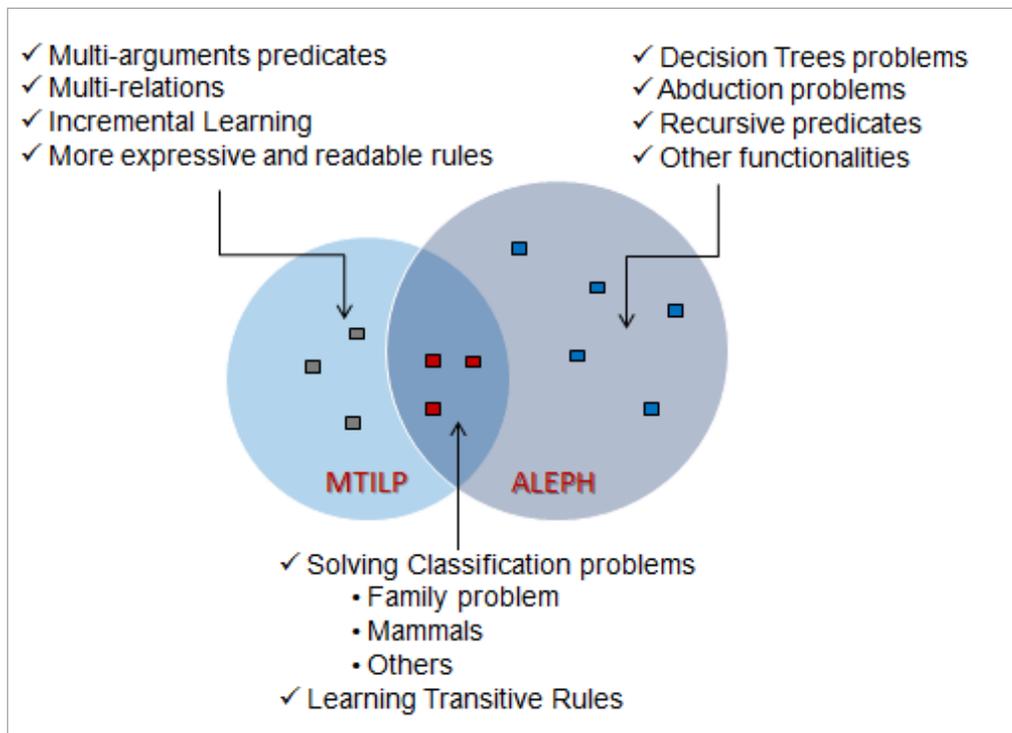


Figure 63: MTILP versus ALEPH summary

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This chapter presents our conclusions and contributions, and then it discusses the limitations of this work. Finally, it concludes with possible directions of future work.

9.1 Major Contributions

In this dissertation we proposed an ILP-based model transformation system that can be used to generate model transformation rules using examples. Two main contributions can be presented here:

- Proposed an ILP-based model transformation system which:
 - employs an ILP system to induce the transformation rules
 - learns the rules from examples consisting of source-target models.
 - requires minimal (less than other MTBE approaches) input to induce the rules from examples, namely the concrete examples without transformation mappings or meta-models.
 - provides the needed tool to convert the given models (in XMI format) into the appropriate format (first order predicates logic) to feed the ILP engine.
 - evaluates the induced rules using a genetic algorithm approach in different ways to find the best rules

- validates the rules against data unseen to ensure their ability in future development.
- produces a rules database that can be utilized, refined and updated by domain experts.
- Identified the limitations of the current ILP system in the context of MDD transformations.
- Proposed the MTILP as a new ILP system that was designed to overcome the limitations of the previous ILP systems. The MTILP system differs from other ILP systems in the following:
 - It allows learning using positive examples only.
 - It does not require modes declarations as a mandatory input to start the induction process.
 - It provides an incremental learning through detecting the correct order of the multiple predicates learning.
 - It uses a genetic algorithm approach to find the best combination of examples during generalization.
 - It provides more expressive and readable rules.

9.2 Limitations and Threats to Validity

There are some limitations to the extent to which these results can be generalized. The following are possible reservations.

The main threat to validity, as with any software engineering research, is scarcity of data and the bias of the datasets selection. Another dimension of scarcity we encountered is

the need to use source/target pairs. It is noteworthy here that different resources have been considered to collect the datasets (student projects, textbooks, reverse engineering). Using different sources helps ensure that the datasets are collected in unbiased manner.

In addition, selecting randomly learning samples that are different from the validation samples would give the results of the experiments some credibility as not being biased. Nevertheless, this does not necessarily mean that the derived transformation rules are complete.

It has been very difficult to find real-world systems with corresponding requirements analysis and initial/detailed software design pairs publically available for us to use for training and validating our rules. It is noteworthy here that training the system using a pair from a source that is different from the source of pairs we used for validation would give the results of the experiments some credibility as not being biased.

Nevertheless, this does not necessarily mean that the derived transformation rules are complete. Moreover, it does not mean that the derived transformation rules for packaging and interfacing can always produce the perfect related artifacts in all cases.

Another threat to validity of this work is the incompleteness in terms of transformation problems coverage. We only considered two major designs activities to show the power of ILP in generalizing rules in MDD context.

9.3 Future Work

Additional research directions that can be explored in future work include:

- Investigate more transformation tasks in the context of analysis-design models transformations.
- Different views of models can be used as input to describe the same system in both source and target aspects.
- Measuring the quality of the target models can be considered in order to provide the user with the best generated model, especially when having more than options when applying the rules. Such situation is called *conflict resolution*.
- More datasets in other contexts can be used to validate MTILP and compare its performance against other ILP systems.

Future effort will try to make contact with some software houses to allow using their repositories and expertise in evolving a generic transformation system. The scalability of the approach will be tested more realistically in this case.

References

- [1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, February 2006.
- [2] R. Soley, " Model driven architecture," Technical report, Object Management Group 2000.
- [3] S. Kent, "Model driven engineering," in *Third International Conference In Integrated formal methods*, Turku, Finland, 2002, pp. 286- 298.
- [4] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*: John Wiley & Sons, 2013.
- [5] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model-Driven Architecture: Practice and Promise*: Addison Wesley, 2003.
- [6] W. Frakes, "Systematic software reuse: a paradigm shift," in *Proceedings of 1994 3rd International Conference on Software Reuse*, 1994, pp. 2-3.
- [7] S. Sadaoui and P. Yin, "Generalization and instantiation for component reuse," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, pp. 175-200, 2006.
- [8] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, pp. 131-183, June 1992.
- [9] B. Coulangue, *Software Reuse*. London: Springer Verlag, 1997.
- [10] S. Russell, P. Norvig, and A. Intelligence, "Artificial Intelligence : A modern approach," Prentice-Hall, Egnlewood Cliffs 25, 1995.
- [11] S. Sendall and W. Kozaczynski, "Model Transformation - The Heart and Soul of Model-Driven Software Development," *IEEE Software, Special Issue on Model Driven Software Development*, pp. 42-45, 2003.
- [12] M. Faunes, H. Sahraoui, and M. Boukadoum, "Generating Model Transformation Rules from Examples Using an Evolutionary Algorithm," in *The 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, 2012, pp. 250-253.
- [13] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, pp. 621-645, 2006.

- [14] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Search-Based Model Transformation by Example," *Software and System Modeling Journal- Special Issue of MODELS08*, 2010.
- [15] F. A. Lisi, "Learning Onto-Relational Rules with Inductive Logic Programming," in *Proceedings of CoRR*, 2012.
- [16] X. Dolques, M. Huchard, C. Nebut, and P. Reitz, "Learning transformation rules from transformation examples: An approach based on Relational Concept Analysis," in *published in EDOC 2010: 14th International Enterprise Distributed Object Computing Conference*, Vittoria : Brazil, 2010.
- [17] D. Zhang and J. J. P. Tsai, "Machine Learning and Software Engineering," *Software Quality Journal*, vol. 11, pp. 87-119, 2003.
- [18] M. A. Ahmed and H. A. Al-Jamimi, "Machine Learning Approaches for Predicting Software Maintainability: A Fuzzy-based Transparent Model," *IET Software* vol. 7, pp. 317-326, 2013.
- [19] H. A. Al-Jamimi and M. Ahmed, "Machine Learning-based Software Quality Prediction Models: State of the Art," in *The 4th International Conference on Information Science and Applications*, June 24-26 Pattaya, Thailand, 2013
- [20] T. Mitchell, *Machine Learning*: McGraw Hill, 1997.
- [21] R. S. Michalski, "A theory and methodology of inductive learning," *Artificial intelligence*, vol. 20, pp. 111-161, 1983.
- [22] S. Muggleton, "Inductive logic programming," *New Generation Computing*, vol. 8, pp. 295-317, 1991.
- [23] S. H. Muggleton, *Inductive Logic Programming*, 1992.
- [24] S. Muggleton and L. D. Raedt., "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19 pp. 629-679, 1994.
- [25] H. A. Al-Jamimi and M. A. Ahmed, "Transition from Analysis to Software Design: A Review and New Perspective " *The International Journal of Soft Computing and Software Engineering* vol. 3, 2013.
- [26] G. Karsai, J. Sztipanovits, Á. Lédeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of IEEE*, vol. 91 pp. 145-164, 2003.

- [27] "Object Management Group, Unified Modeling Language: Infrastructure Version 2.1.1," 2007
- [28] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation and Its Application to Graph Transformation," in Proceedings of the International Workshop on Graph and Model Transformation, Tallinn, Estonia, 2005, pp. 7-23.
- [29] D. Varró, "Model Transformation by Example," in In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, 2006, pp. 410-424.
- [30] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, "Towards Model Transformation Generation By-Example," in In: 40th Hawaiian Int. Conf. on Systems Science (HICSS 2007), 2007.
- [31] H. Lieberman, "Your wish is my command: programming by example," Morgan Kaufmann Publishers Inc. , 2001.
- [32] P. Langer, M. Wimmer, and G. Kappel, "Model-to-Model Transformations By Demonstration," in In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, 2010, pp. 153-167.
- [33] Y. Sun, J. White, and J. Gray, "Model Transformation by Demonstration," in In: Schurr, A., Selic, B. (eds.) MODELS 2009. LNCS, 2009, pp. 712-726.
- [34] I. Bratko and S. Muggleton, "Applications of inductive logic programming," Communications of the ACM, vol. 38, pp. 65-70, 1995.
- [35] T. M. Mitchell, "Generalization as search," Artificial intelligence vol. 18, pp. 203-226, 1982.
- [36] A. Srinivasan, "The Aleph Manual," University of Oxford, 2007.
- [37] J. Lloyd, Foundations of Logic Programming: Berlin: Springer-Verlag, 1987.
- [38] J. W. Lloyd and W. T. Rodney, "Making Prolog more expressive," The Journal of Logic Programming, vol. 1, pp. 225-240, 1984.
- [39] N. Lavrac and S. Dzeroski, Inductive Logic Programming: Techniques and Applications: Ellis Horwood, 1994.
- [40] L. D. Raedt, N. Lavrac, and S. Dzeroski, "Multiple predicate learning," in In Proceedings of the 13th international joint conference on artificial intelligence, 1993, pp. 1037-1042.
- [41] S. Muggleton and C. Feng, "Efficient induction of logic programs," Inductive logic programming, vol. 38 pp. 281-298, 1992.

- [42] L. Raedt and N. Lavrac, "Multiple predicate learning in two ILP settings," *Journal of the IGPL*, vol. 4, pp. 227-254, 1996.
- [43] J. R. Quinlan, "Learning logical definitions from relations," *Machine Learning*, vol. 5, pp. 239-266, 1990.
- [44] S. Muggleton, "Inverse entailment and Progol," *New generation computing*, vol. 13, pp. 245-286, 1995.
- [45] S. Muggleton and A. Tamaddoni-Nezhad, "QG/GA: a stochastic search for Progol," *Machine Learning*, vol. 70, pp. 121-133, 2008.
- [46] S. Muggleton and B. Wray, "Machine invention of first-order predicates by inverting resolution," in *Proceedings of the fifth international conference on machine learning*, 1992, pp. 339-352.
- [47] M. d. C. Nicoletti, F. O. d. S. Lisboa, and E. R. H. Jr, "Learning temporal interval relations using inductive logic programming," in *In Integrated Computing Technology: Springer Berlin Heidelberg*, 2011, pp. 90-104.
- [48] S. Muggleton, "Inductive logic programming: issues, results and the challenge of learning language in logic," *Artificial Intelligence* vol. 114, pp. 283-296, 1999.
- [49] S. Muggleton, L. D. Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan, "ILP turns 20," *Machine Learning*, vol. 86, pp. 3-23, 2012.
- [50] R. R. Trippi and J. K. Lee, *Artificial intelligence in finance & investing: Chicago, IL, Irwin*, 1996.
- [51] P. R. Cohen and E. A. Feigenbaum, *The handbook of artificial intelligence* vol. 3: Butterworth-Heinemann, 2014.
- [52] E. Friedman-Hill, *Jess in Action: Java Rule-Based Systems: Manning Publications*, 2003.
- [53] A. Montes, H. Pacheco, H. Estrada, and O. Pastor, "Conceptual Model Generation from Requirements Model: A Natural Language Processing Approach," *L. V. 5039, Ed.: Springer*, 2008, pp. 325-326.
- [54] I. Diaz, L. Moreno, I. Fuentes, and O. Pastor, "Integrating Natural Language Techniques in OO-Methods," in *Computational Linguistics and Intelligent Text Processing, LNCS. vol. 3406: Springer* 2005, pp. 177-188.

- [55] T. Yue, L. C. Briand, and Y. Labiche, "An Automated Approach to Transform Use Cases into Activity Diagrams," *Modelling Foundations and Applications*, LNCS, vol. 6138, pp. 337-353, 2010.
- [56] S. Kalaivani, L. Dong, H. Behrouz, and A. Eberlein, "UCDA: Use Case Driven Development Assistant Tool for Class Model Generation," in *SEKE*, Banff, Canada, 2004, pp. 324-329.
- [57] S. P. Overmyer, L. Benoit, and R. Owen, "Conceptual modeling through linguistic analysis using LIDA," in *Proc. of the 23rd Int. Conf. of Software Engineering (ICSE)*, Toronto, Canada, 2001, pp. 401-410.
- [58] D. K. Deeptimahanti and R. Sanyal, "An Innovative Approach for Generating Static UML Models from Natural Language Requirements," *Advances in Software Engineering, Communications in Computer and Information Science*, vol. 30, pp. 147-163, 2009.
- [59] D. D. Kumar and R. Sanyal, "Static UML Model Generator from Analysis of Requirements (SUGAR)," in *International Conference on Advanced Software Engineering and Its Applications (ASEA 2008)*, 2008, pp. 77-84.
- [60] D. K. Deeptimahanti and M. A. Babar, "An Automated Tool for Generating UML Models from Natural Language Requirements.," in *IEEE / ACM Int. Conf. on ASE*, 2009.
- [61] E. Insfrán, O. Pastor, and R. Wieringa, "Requirements Engineering-Based Conceptual Modeling," *Requirements Engineering Journal*, vol. 7, pp. 61-72, 2002.
- [62] P. More and R. Phalnikar, "Generating UML Diagrams from Natural Language Specifications," *International Journal of Applied Information Systems*, vol. 8, 2012.
- [63] I. S. Bajwa, A. Samad, and S. Mumtaz, "Object Oriented Software Modeling Using NLP Based Knowledge Extraction," *European Journal of Scientific Research* ISSN 1450-216X, vol. 35 pp. 22-33 2009.
- [64] M. Ibrahim and R. Ahmad, "Class diagram extraction from textual requirements using Natural language processing (NLP) techniques," in *Proceedings of the Second International Conference on Computer Research and Development.*, 2010.
- [65] S. K. Shinde, V. Bhojane, and P. Mahajan, "NLP based Object Oriented Analysis and Design from Requirement Specification," *International Journal of Computer Applications* (0975 - 8887), vol. 47, June 2012.

- [66] M. Y. Santos and Ricardo J. Machado "On the Derivation of Class Diagrams from Use Cases and Logical Software Architectures," in Fifth International Conference on Software Engineering Advances, 2010
- [67] R. Giganto and T. Smith, "Derivation of Classes from Use Cases Automatically Generated by a Three-Level Sentence Processing Algorithm," in Proc. of the Third International Conference on Systems IEEE Computer Society, 2008.
- [68] S. Sarkar, V. S. Sharma, and R. Agarwal, "Creating Design from Requirements and Use Cases: Bridging the Gap between Requirement and Detailed Design," in Proceedings of ISEC '12, Feb. 22-25 Kanpur, UP, India, 2012
- [69] T. Yue, L. C. Briand, and Y. Labiche, "Automatically Deriving a UML Analysis Model from a Use Case Model," 2010.
- [70] T. Yue, L. Briand, and Y. Labiche, "Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments," 2010.
- [71] T. Yue, S. Ali, and L. Briand, "Automated Transition from Use Cases to UML State Machines to Support State-Based Testing," in In: ECMFA, 2011, pp. 115-131.
- [72] P. Grunbacher, A. Egyed, and N. Medvidovic, "Reconciling software requirements and architectures with intermediate models," *Softw. Syst. Model*, vol. 3, pp. 235-253, 2003.
- [73] D. Liu and H. Mei, "Mapping requirements to software architecture by feature-orientation," *Requirements Engineering Journal*, vol. 25, pp. 69-76, 2003.
- [74] H. Kaindl and J. Falb, "Can We Transform Requirements into Architecture?," in in 3rd Int'l Conf. on Software Engineering Advances, 2008, pp. 91-96.
- [75] C. Larman, *Applying UML and Patterns*: .Prentice Hall PTR, 2004.
- [76] G. Ebner and H. Kaindl, "Tracing all around in reengineering," *IEEE Software*, pp. 70-77, 2002
- [77] H. Kaindl and J. Falb, "From Requirements to Design: Model-driven Transformation or Mapping," in in Proc. First International Workshop on Model Reuse Strategies (MoRSe 2006), , Warsaw, Poland, 2006, pp. 29-32.
- [78] J. e. L. Garridoa, M. Nogueraa, M. Gonz´alezb, M. V. Hurtado, and M. L. Rodríguez, "Definition and use of Computation Independent Models in an MDA-based groupware development process," *Science of Computer Programming* vol. 66 pp. 25-43, 2007.

- [79] M. Kardoš and M. Drozdová, "Analytical method of CIM to PIM transformation in Model Driven Architecture (MDA)," *Journal of Information and Organizational Sciences* vol. 34, pp. 89-99, 2010.
- [80] W. Zhang, H. Mei, H. Zhao, and J. Yang, "Transformation from CIM to PIM: A Feature-Oriented Component-Based approach," in *MoDELS*, Montego Bay, Jamaica, 2005.
- [81] S. Kherraf, É. Lefebvre, and W. Suryn, "Transformation from CIM to PIM using patterns and Archetypes," in *19th Australian Conference on Software Engineering*, 2008, pp. 338-346.
- [82] X. Cao, H. Miao, and Y. Chen, "Transformation from computation independent model to platform independent model with pattern," *Journal of Shanghai University (English Edition)* vol. 12 pp. 515-523, 2008.
- [83] V. D. Castro, E. Marcos, and J. M. Vara, "Applying CIM-to-PIM model transformations for the service-oriented development of information systems," *Information and Software Technology*, vol. 53 pp. 87-105, 2011.
- [84] A. Rodríguez, I. G.-R. d. Guzmán, E. Fernández-Medina, and M. Piattini, "Semi-formal transformation of secure business processes into analysis class and use case models: An MDA approach," *Information and Software Technology*, vol. 52 pp. 945-971, 2010.
- [85] A. Raj, T. V. Prabhakar, and S. Hendryx, "Transformation of SBVR Business Design to UML Models," in *ISEC'08*, February 19-22, Hyderabad, India, 2008.
- [86] E. Suarez, M. Delgado, and E. Vida, "Transformation of a Process Business Model to Domain Model," in *Proceedings of the World Congress on Engineering 2008 Vol I, WCE 2008*, July 2 - 4, London, U.K., 2008.
- [87] I. Oliver, "Model based testing and refinement in MDA based development," in *Advances in Design and Specification Languages for SoCs*, 2005, pp. 107-122.
- [88] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation," in *Proc. Intl. Workshop on Graph and Model Transformation*, 2003.
- [89] M. Biehl, "Literature Study on Model Transformations," Jul. 2010.
- [90] F. Jouault, F. Allilaire, J. Bezivin, and I. Kurtev, "ATL: a model transformation tool," *Science of Computer Programming*, vol. 72 pp. 31-39, 2008.

- [91] I. Kurtev, "State of the art of QVT: A model transformation language standard " in Applications of graph transformations with industrial relevance: Springer Berlin Heidelberg, 2008, pp. 377-393.
- [92] M. Lawley and J. Steel, "Practical Declarative Model Transformation with Tefkat," in LECTURE NOTES IN COMPUTER SCIENCE, 2006, pp. 139-150.
- [93] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Model Transformation as an Optimization Problem," in In Proceedings of the 11th international Conference on Model Driven Engineering Languages and Systems (MODELS08), 2008, pp. 159-173.
- [94] M. Faunes, H. Sahraoui, and M. Boukadoum, "Genetic-programming approach to learn model transformation rules from examples," in Theory and Practice of Model Transformations: Springer Berlin Heidelberg, 2013, pp. 17-32.
- [95] Z. Balogh and D. Varró, "Model transformation by example using inductive logic programming," Software and Systems Modeling, vol. 8, pp. 347-364, 2009.
- [96] M. Strommer and M. Wimmer, "A Framework for Model Transformation By-Example: Concepts and Tool Support," Objects, Components, Models and Patterns Lecture Notes in Business Information Processing, vol. 11, pp. 372-391 2008.
- [97] X. Dolques, M. Huchard, and C. Nebut, "From transformation traces to transformation rules: Assisting Model Driven Engineering approach with Formal Concept Analysis," in 17th International Conference on Conceptual Structures (ICCS 2009), Moscow, Russia, 2009, pp. 15-29.
- [98] X. Dolques, A. Dogui, J.-R. Falleri, M. Huchard, C. Nebut, and F. Pfister, "Easing Model Transformation Learning with Automatically Aligned Examples," in Modelling Foundations and Applications, Birmingham, UK 2011, pp. 189-204.
- [99] N. F. Noy and M. A. Musen, "Anchor-prompt: Using non-local context for semantic matching," in In Proceedings of the workshop on ontologies and information sharing at the international joint conference on artificial intelligence, Seattle, USA, 2001, pp. 63-70
- [100] M. Huchard, M. R. Hacène, C. Roume, and P. Valtchev, "Relational concept discovery in structured datasets," Annals of Mathematics and Artificial Intelligence, vol. 49, pp. 39-76, 2007.

- [101] H. Saada, X. Dolques, M. Huchard, C. Nebut, and H. Sahraoui, "Generation of Operational Transformation Rules from Examples of Model Transformations," in Model Driven Engineering Languages and Systems. vol. 7590, 2012, pp. 546-561.
- [102] "Jess rule engine <http://herzberg.ca.sandia.gov/jess>."
- [103] H. Saada, X. Dolques, M. Huchard, C. Nebut, and H. Sahraoui, "Learning Model Transformations from Examples using FCA: One for All or All for One?," in CLA'2012: 9th International Conference on Concept Lattices and Applications, Fuengirola (Málaga) : Espagne (2012), 2012.
- [104] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, Genetic Programming: An Introduction. San Mateo, CA: Morgan Kaufmann, 1998.
- [105] I. García-Magariño, J. J. G. Sanz, and R. Fuentes-Fernández, "Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages," in In: Paige, R.F. (ed.) ICMT 2009. LNCS, 2009, pp. 52-66.
- [106] F. Budinsky, Eclipse Modelling Framework: Developer's Guide: Addison Wesley, 2003.
- [107] J. B. Warmer and A. G. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA: Addison-Wesley Professional, 2003.
- [108] Y. Sun, "Supporting Model Evolution through Demonstration-based Model Transformation," Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09), pp. 779-780, 2009.
- [109] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger., "An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example," in In Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2009, pp. 271-285.
- [110] M. Kessentini, A. Bouchoucha, H. Sahraoui, and M. Boukadoum, "Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search," in In Proceedings of the Sixth European Conference on Modelling Foundations and Applications ECMFA2010, 2010.
- [111] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating Transformation Rules from Examples for Behavioral Models," in In Proceedings of Behavioural Modelling - Foundations and Application (BM-FA 2010), 2010.

- [112] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in In: Proceedings of IEEE International Conference on Neural Networks, 1995, pp. 1942-1948.
- [113] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Sciences* vol. 220, pp. 671-680, 1983.
- [114] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object oriented software," in proceedings of the 3rd Working Conference on Reverse Engineering., 1996, pp. 208-216.
- [115] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. New York, NY: Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, 1995.
- [116] R. Wuyts, "Declarative reasoning about the structure of object oriented systems," in In proceedings of the 26th conference on the Technology of Object-Oriented Languages and Systems: IEEE Computer Society Press, 1998, pp. 112-124.
- [117] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating Design Pattern Detectors from Pattern Specifications," in In Proc. of the 18th IEEE International Conference on Automated Software Engineering Montreal, Quebec, Canada: IEEE Computer Society Press, 2003, pp. 245-248.
- [118] H. Huang, S. Zhang, J. Cao, and Y. Duan, "A practical pattern recovery approach based on both structural and behavioral analysis," *Journal of Systems and Software*, vol. 75, pp. 69-87, 2005.
- [119] A. Stoianov and I. Sora, "Detecting Patterns and Antipatterns in Software Using Prolog Rules," *Proceedings of International Joint Conference on Computational Cybernetics and Technical Informatics*, pp. 253-258, 2010.
- [120] "ILP Applications Descriptions <http://www-ai.ijs.si/~ilpnet2/apps/index.html>." vol. 2013.
- [121] W.-J. Hou and H.-Y. Chen, "Rule Extraction in Gene-Disease Relationship Discovery," *Gene* 2012.
- [122] Y. Qiu, K. Shimada, N. Hiraoka, K. Maeshiro, W.-K. Ching, K. F. Aoki-Kinoshita, and K. Furuta, "Knowledge discovery for pancreatic cancer using inductive logic programming," *IET systems biology*, vol. 8, pp. 162-168, 2014.
- [123] T. O. D. Beéck, A. Hommersom, J. V. Haaren, M. v. d. Heijden, L. O. Jesse Davis, and I. Nagtegaal, "Mining hierarchical pathology data using inductive logic programming," in *Proceedings of the 15th Conference of Artificial Intelligence in Medicine*, 2015.

- [124] R. King, M. Sternberg, and A. Srinivasan., "Relating chemical activity to structure: An examination of ilp successes," *New Generation Computing*, vol. 13, 1995.
- [125] Houssam Nassif, H. Al-Ali, S. Khuri, W. Keirouz, and D. Page, "An inductive logic programming approach to validate Hexose binding biochemical knowledge," in *In Inductive Logic Programming: Springer Berlin Heidelberg*, 2010, pp. 149-165.
- [126] J. C. Santos, H. Nassif, D. Page, S. H. Muggleton, and M. J. Sternberg, "Automated identification of protein-ligand interaction features using inductive logic programming: A hexose binding case study," *BMC bioinformatics* vol. 13, p. 162, 2012.
- [127] T. Luu, N. Nguyen, A. Friedrich, J. Muller, L. Moulinier, and O. Poch, "Extracting knowledge from a mutation database related to human monogenic disease using inductive logic programming," in *International Conference on Bioscience, Biochemistry and Bioinformatics*, pp. 83-100, 2011.
- [128] Fumio Mizoguchi, H. Ohwada, H. Nishiyama, and H. Iwasaki, "Identifying Driver's Cognitive Load Using Inductive Logic Programming," in *Inductive Logic Programming- Springer Berlin Heidelberg*, 2013, pp. 166-177.
- [129] J. Cussens, "Part-of-speech disambiguation using ilp," Technical report, Oxford University Computing Laboratory 1996.
- [130] T. Horvth, G. Paass, F. Reichartz, and S. Wrobel, "A logic-based approach to relation extraction from texts," in *Inductive Logic Programming- Springer Berlin Heidelberg*, pp. 34-48, 2010.
- [131] S. Thaicharoen, T. Altman, K. Gardiner, and K. J. Cios, "Discovering relational knowledge from two disjoint sets of literatures using inductive logic programming," in *In Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE Symposium on*, 2009, pp. 283-290.
- [132] R. Lima, B. Espinasse, H. Oliveira, L. Pentagrossa, and F. Freitas, "Information Extraction from the Web: An Ontology-based Method using Inductive Logic Programming," in *IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2013, pp. 741-748.
- [133] S. Džeroski, N. Jacobs, M. Molina, C. Moure, S. Muggleton, and W. V. Laer, "Detecting traffic problems with ILP," *Springer Berlin Heidelberg*, 1998.

- [134] G. Leban, J. Žabkar, and I. Bratko, "An experiment in robot discovery with ILP," Springer Berlin Heidelberg, pp. 77-90, 2008.
- [135] N. C. M. K. H. F. R. Hasegawa, "Rule Extraction from Blog Using Inductive Logic Programming," in In 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). vol. 3, 2010, pp. 269-272.
- [136] N. Chikara, M. Koshimura, H. Fujita, and R. Hasegawa, "Rule Extraction from Micro-Blog Using Inductive Logic Programming," in In 2012 Spring Congress on Engineering and Technology (S-CET), 2012, pp. 1-4.
- [137] Y. Dehbi and L. Plümer, "Learning grammar rules of building parts from precise models and noisy observations," ISPRS Journal of Photogrammetry and Remote Sensing, vol. 66, pp. 166-176, 2011.
- [138] M. Bayoudh, E. Roux, R. Nock, and G. Richard, "Automatic learning of structural knowledge from geographic information for updating land cover maps," in In Symposium of the Latin American Society for Remote Sensing and Spatial Information Systems (SELPER), 2012.
- [139] R. Lima, B. Espinasse, H. Oliveira, R. Ferreira, L. Cabral, F. Freitas, and R. Gadelha, "An Inductive Logic Programming-Based Approach for Ontology Population from the Web," in In Database and Expert Systems Applications: Springer Berlin Heidelberg, 2013, pp. 319-326.
- [140] F. Lisi and U. Straccia, "An Inductive Logic Programming Approach to Learning Inclusion Axioms in Fuzzy Description Logics," in In CILC, 2011, pp. 57-71.
- [141] H. Nguyen, T.-D. Luu, O. Poch, and J. D. Thompson, "Knowledge Discovery in Variant Databases Using Inductive Logic Programming," Bioinformatics and biology insights, vol. 7, 2013.
- [142] M. Essaidi, A. Osmani, and C. Rouveirol, "Transformation Learning in the Context of Model-Driven Data Warehouse: An Experimental Design Based on Inductive Logic Programming," in The 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI): IEEE, 2011, pp. 693-700.
- [143] D. Varró and Z. Balogh, "Automating Model Transformation by Example Using Inductive Logic Programming," in SAC'07, Seoul, Korea, 2007.
- [144] S. Sankaranarayanan, F. Ivancic, and A. Gupta, "Mining library specifications using inductive logic programming," in ACM/IEEE 30th International Conference on Software Engineering, 2008.(ICSE'08). , Leipzig, Germany, 2008.

- [145] G. Contissa and M. Laukyte, "Legal Knowledge Representation: A Twofold Experience in the Domain of Intellectual Property Law," in In the Twent-first annual conference on Legal Knowledge and Information Systems 2008, pp. 98-107.
- [146] R. Briggs, "Knowledge representation in Sanskrit and artificial intelligence," AI magazine, vol. 6, 1985.
- [147] H. A. Al-Jamimi and M. Ahmed, "Using Prolog Rules to Detect Software Design Patterns: Strengths and Weaknesses," in The Twenty-Fifth International Conference on Software Engineering and Knowledge Engineering (SEKE 2013), Boston, USA, 2013.
- [148] S. Muggleton, "Learning from positive data," in In Inductive logic programming, 1997, pp. 358-376.
- [149] A. Jorge and P. B. Brazdil, "Integrity constraints in ILP using a Monte Carlo approach," Inductive Logic Programming. Springer Berlin Heidelberg, pp. 229-244, 1997.
- [150] M. O'docherty, "Object-Oriented Analysis & Design," John Wiley & Sons, 2005.
- [151] "General Inductive Logic Programming System, <http://www.doc.ic.ac.uk/~jcs06/GILPS/>."

Vitae

Name :{ Hamdi Ali Ahmed Al-Jamimi }

Nationality :{ Yemeni }

Date of Birth :{12/28/1978}

Email :{ aljamimih@gmail.com }

Address :{ Yemen – Sana’a }

Academic Background | Hamdi Al-Jamimi earned his BS degree with honors in Computer Science from Tamar University, Yemen in July 2001. He obtained his MS in Computer Science from King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia in June 2010. Prior to attending KFUPM, he served for four years as a full time lecturer in Tamar University. Al-Jamimi got the opportunity to pursue his PhD studies in KFUPM as a full time student. He completed his PhD in Computer Science and Engineering in May 2015. He is specialized in Software Engineering (SWE) and it's synergy with diverse research areas including search-based SWE, model driven engineering, software refactoring, and object-oriented analysis and design. His entire research career target the intersection between SWE and machine learning through the applications of computational intelligence in SWE. During his studies at KFUPM he has published his research results in various international journals and conferences. Al-Jamimi is a member of the IEEE and has served as a technical program committee member and a reviewer of many international conferences and journals.]