Model Driven Development Transformations using Inductive Logic Programming

Hamdi A. Al-Jamimi and Moataz A. Ahmed
Information and Computer Science Department
King Fahd University of Petroleum and Minerals,
Dhahran, 31261, Kingdom of Saudi Arabia

***Abstract*—Model transformation by example is a novel approach in model-driven software engineering. The rationale behind the approach is to derive transformation rules from an initial set of interrelated source and target models; e.g., requirements analysis and software design models. The derived rules describe different transformation steps in a purely declarative way. Inductive Logic Programming utilizes the power of machine learning and the capability of logic programming to induce valid hypotheses from given examples. In this paper, we use Inductive Logic Programming to derive transformation rules from given examples of analysis-design pairs. As a proof concept, we applied the approach to two major software design tasks: class packaging and introducing Façade design. Various analysis-design model pairs collected from different sources were used as case studies. The resultant performance measures show that the approach is promising.**

Keywords—Transformation model; software design models; transformation rules; inductive logic programming

I. Introduction

The problem of transforming the requirement analysis models into software design models can be viewed as a model transformation problem. Designers utilize their engineering knowledge to perform this specific kind of transformation. In this paper, we capture such knowledge through learning the transformation rules from available pairs of requirement analysis models (e.g., domain model or conceptual class diagram) and corresponding software design models (e.g., component diagram or package diagram). The approach of learning a model transformation from provided examples is referred to as Model Transformation by Example (MTBE) [1]. The examples, in this context, represent pairs of the transformation requirements/design models.

Model Driven Development (MDD) considers a sequence of several kinds of models as the primary artifacts of the development process as they contain the needed information that supports its different phases. Those models may be derived from each other via automated transformation. The models are structured conforming to particular models called meta-models. Implementing models transformation requires an intense knowledge about MDD including the meta-models and the environment of the model transformation.

Practically, machine learning (ML) techniques can be used to deduce the transformation rules from the available set of examples [2]. ML techniques have been applied in different domains, including software engineering [3]-[8]. Inductive logic programming (ILP) is one of the machine learning techniques that provide mechanisms for inducing valid hypotheses from given examples and background knowledge of the domain of interest [9], [10]. Rules have been used widely as a powerful way for representing knowledge. However, in the domain of model transformation, authoring the transformation rules is not a trivial task. It might be easier for the domain expert to provide examples of the transformation rather than introducing consistent and complete rules. Thus, it is desirable to utilize the accumulated experience by automatically capturing the transformation rules from examples [11].

The primary contribution of this paper is to define a methodology, with an associated tools-chain, for the incremental design of model transformation rules. The increments in the rule design are automatically derived by defining positive and negative examples on a given training set of models (i.e. learning models). As a secondary contribution, we propose the application of such an inferred set of model transformation rules in order to refactor actual domain/conceptual artifacts (referred as "analysis models") toward design solutions. These contributions are interesting for both the communities working on Model Driven Engineering and on Software Engineering, and the context defined here with their solution may be worth for the attention also to a wider audience from others communities.

In a particular, we use ILP to automatically capture the expertise manifested in previous analysis-design pairs, and consequently, represent such expertise in a form of declarative rules. These rules can be applied to a new design problem to suggest a possible design to given analysis models. Such design suggestion can be adopted "as is" by the designer or at least be reviewed and refined by the designer before adoption. In either cases, this would offer effort saving and, accordingly, cost reduction. Moreover, this would offer indirect reuse of best practice that would in turn improve quality. We applied the approach to various case studies collected from different sources. A considerable part of the data have been used for training to induce rules regarding two major software design tasks: class packaging and introducing Façade design.

The rest of this paper is organized as follows. Section II introduces the needed technical background. Section III reviews the literature survey, while Section IV introduces the proposed transformation system. Section V describes the transformations tasks. While Section VI demonstrates the conducted experiments, Section VII discusses the findings and the open issues. Finally, Section VIII concludes the paper.

II. Technical Background

In this section, we give the background necessary to follow the rest of the paper.

A. *Model Driven Development Transformations*

The goal of this work is to facilitate the transformation from the analysis models toward the software design models by reusing previous experience. That is, based on the given requirements, the existing requirement-design pairs from previous systems can be utilized to build the new system's design. Indeed rule-based transformation approaches rely on transformation rules that were obtained empirically [12]. However, it might be a difficult task to define, express, and maintain the transformation rules, particularly for non-widely used formalisms. That is, it is most important to gather the knowledge in a form of rules, not only to decide about the transformation language [13], [14]. Thus, the objective of this paper is to use ILP, discussed next, to capture such transformation rules.

B. *Inductive Logic Programming*

ILP can be seen as the intersection of machine learning and logic programming. An ILP problem is defined as follows: Given a background theory B, and a set of examples E (represented as ground literals) that consists of positive E+ and negative E- examples, the target is to find a hypothesis H such that $\forall e^{+} \in E^{+} : B \cup H \models e$ and $\forall e^{-} \in E^{-} : B \cup H \neq e^{-}$ [9], [10]. Thus, the problem of learning a particular hypothesis can be designed as a search problem through a space of models [15]. To perform a search two main strategies were used: generate-and-test and data-driven. In both, the applied algorithms can proceed either bottom-up or top-down. A combination of those strategies and algorithms can be exploited. Examples of ILP systems are FOIL [16], GOLEM [17], PROGOL [18], ALEPH [19] and others. ALEPH (A Learning Engine for Proposing Hypotheses) employed in this work, has different evaluation functions and search strategies that can be applied, and it has been applied successfully in many domains [20]-[25].

III. Literature Survey

In this section, we present a literature survey that addresses two views presented below. 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. In the rest of the paper, we use the term transformation mapping.

A. *Model Transformation by Examples Approaches*

MTBE approach has been initiated by Varró [1], where he derives the transformation rules from an initial set of examples that includes interrelated source and target models. The user provides the examples, and then the developers refine the derived rules. The transformation rules are produced using an ad-hoc algorithm by utilizing transformation mapping and corresponding meta-models. Balogh and Varró [21] improve the original work of Varró by using ILP instead of the original ad-hoc heuristic. Nevertheless, a semi-automatic process needs interacting with the ILP inference engine and requires detailed transformation mappings. Wimmer et al. [26] generates ATL (ATLAS Transformation Language) rules [27] with using transformation mappings to assist the derivation of model transformation rules. Dolques et al. [28] use Relational Concept Analysis [29] to derive commonalities between the source and target meta-models and transformation mappings. However, the transformation patterns cannot be executed directly. This approach was extended by Saada et al. [30] to learn transformation patterns from the examples, then those patterns are analyzed, filtered and transformed into operational transformation rules. Some MTBE approaches generate n-to-m transformation rules. In [31], [32], the rules are generated from meta-models to satisfy some developer constraints. ATL has been used to implement the generated rules. Another many-to-many rules generator proposed by Faunes et al. [13]. They adapted genetic programming to generate transformation rules expressed in Jess, a fact-based rule language. Jess[1] is a tool for building a type of intelligent software i.e., expert systems.

In conclusion, the conducted survey revealed that most of the approaches that derive transformation rules use transformation example pairs and, with the exception of one work [13], all of them use transformation mappings. In addition, to the best of our knowledge none of the current approaches address the problem of transforming requirement analysis models into software design models. Moreover, most of MTBE approaches require the source, target models and their meta-models as well as the detailed mapping between these models to derive the transformation rules. Unlike these MTBE approaches, our approach aims to use the minimal inputs, the source and target models only, to derive the transformation rules. The most similar work to ours is [21]; however they differ in many facets. First, they considered the problem of transforming class diagrams into relational schema models that is different from our problem of transforming analysis models into design models. Second, they also used the connectivity analysis that is considered as transformation mappings at the meta-model level. In contrast, the approach presented in this paper requires only the concrete models without meta-models or connectivity analysis.

B. *Using Inductive Logic Programming to Generate Rules*

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, according to the given data, general rules are generated and 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 [33]. In particular, ILP has been used in solving software engineering problems [21], [34].

IV. ILP-Based Transformation System

Our proposed transformation system comprises of three main components. Fig. 1 demonstrates the system's components and other supporting functions. It is a generic

[1] http://www.jessrules.com/jess/index.shtml

structure where different ILP systems can be employed to induce rules [34].

ILP systems often start with a preliminary pre-processing stage and ends with a post-processing stage [35]. ALEPH requires that the given information should be in the form of clauses. Thus, the preprocessing step in our transformation system focuses on the conversion of the UML models (given in XMI format) into first order logic predicates. XMI stands for XML (Extensible Markup Language) Metadata Interchange. It is an Object Management Group standard for exchanging metadata information via XML. XMI is considered as the de-facto standard format used commonly as an interchange format for UML models.

TABLE I. defines a set of predicates used to represent the UML models artifacts. In our work, each used example pair consists of two UML models: source and target. The former is translated to be the background knowledge, whereas the latter is used to present the positive examples. The given UML models have no negative examples. In such cases, Closed World Assumption (CWA) [36] is used to generate the negative examples. An intermediate step between the rules generation and rule application is considered to translate the rules generalized by ALEPH into fact-based rule language. Finally, the post-processing stage concentrates on improving the efficiency by removing the redundant clauses in the induced theory.

A. *Transformation Rules Generation and Generalization*

Generally, the transformation problem needs a set of transformation rules in order to cover all the aspects in the transformation problem. The transformation rule here is used to analyze a particular aspect of the analysis requirements given as input and synthesize the corresponding software design to be presented as output. In this context, the transformation system can be encoded as a set of transformation rules R={r_1, r_2, …, r_n}. Each rule can be encoded as a pair of promise and conclusion r_i={P, C} where P is the analysis artifacts to search for in the source model and C is the design artifacts to instantiate when producing the target model.

Algorithm 1 demonstrates the steps we follow to generate the transformation rules. Using ALEPH system, an independent run is performed to produce hypothesis or more for a single predicate from the given examples with background knowledge. To run the ALEPH system, there is a need to feed three files containing the knowledge background, the positive and the negative examples. What is significant limitation in most of the current ILP systems is the need to predefine the target predicate before starting the learning process. Two parts included in the background knowledge rules structure and artifacts descriptions. The former guides the construction of a single rule, while the latter describes source models artifacts. Although the same knowledge background file can be used in different runs, the modes (*modeh* and *modeb*) declarations need to be adjusted to help determine what type of rules to learn. While *modeh* describes the head of the target hypothesis, *modeb* describes the atoms expected to appear in the target body. TABLE II. shows examples of two different inputs.

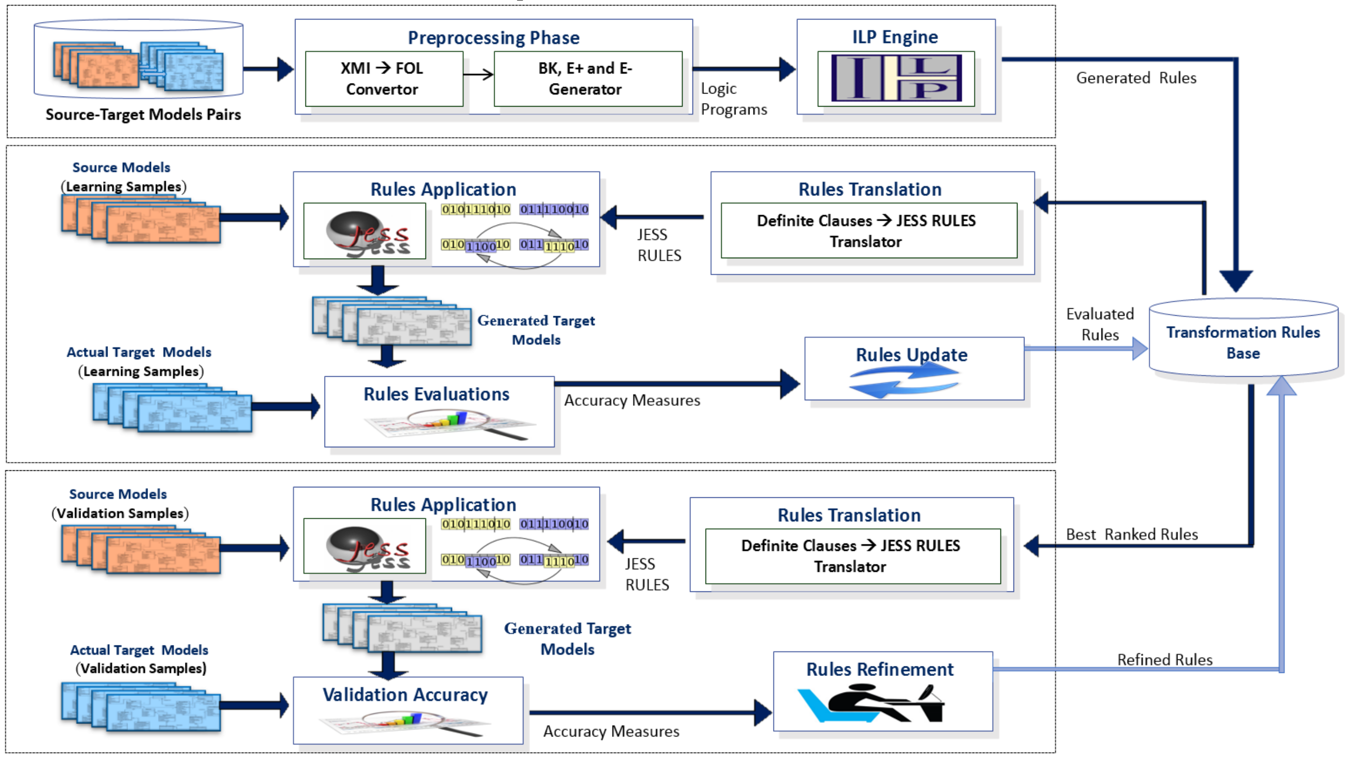


Fig. 1. Architecture of the proposed transformation system.

TABLE I. PREDICATES FOR THE BACKGROUND KNOWLEDGE REPRESENTATION

Predicate	Meaning
package(p)	It defines p as a package.
class(c)	It defines c as a class.
packagehasClass (p, c)	Class c is located into a package p.
classhasOperation(c, op)	Class c has an operation op.
classhasAttribute(c, a)	Class c has an attribute a.
inheritance(c1, c2)	Class c2 is a subclass of class c1.
association(c1, c2)	Class c1 keeps a reference to class c2 where both classes are located in same package.
associationAcrossPackages (p1, c1, p2, c2)	Class c1 keeps a reference to class c2 where both classes are located in different packages p1 and p2.
packageOfClasses({set of classes})	A set consists of one or more class classes located in one package.
interface(p, f)	A package p has an interface f that links the classes located in package p to the classes placed outside of package p to minimize the external relations.

B. Rules Translation and Application

All the induced rules are initially stored in the rule base (RB) in logic programs form. These are then translated into jess script. When a new instance of source model (i.e. requirement analysis) is presented, the models are converted to logic program.

```
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:          $B \leftarrow B \cup s_i$        $\forall s \in S$
6:          $E \leftarrow E \cup t_i$        $\forall t \in T$
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
                 $pe_i = pe_i \cup e_j$
                 $E = E \setminus e_1$
17:      end for
18:  $PE = PE \cup pe_i$
19:  $i = i + 1$
20:  return $B$ and $PE = \{pe_i, \dots, pe_n\}$ $(1 \leq n \geq E)$
```

Applying the translated rules on the new source model means that rule(s) might fire when some facts satisfy its conditions. Firing a rule means some facts are asserted or some others may be retracted. The obtained facts, after application, are supposed to represent the corresponding software design, shown in Algorithm 2.

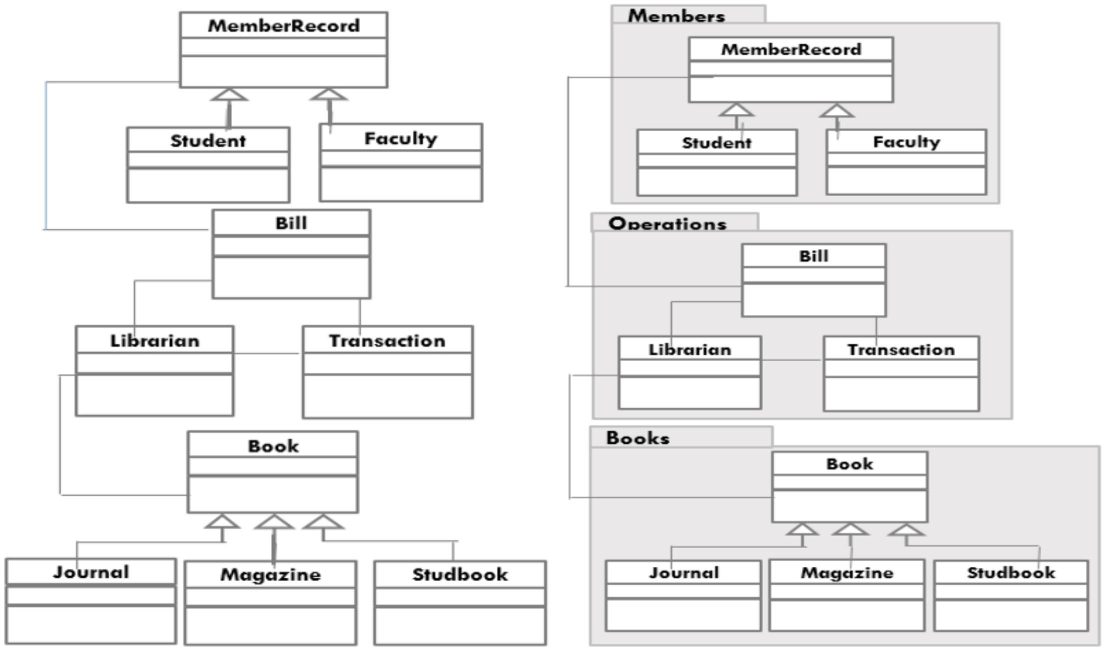
```
Algorithm 2: Transformation Rules Generation

$ST = \{S,T\}$ : Pairs of source and target models
$K$: Knowledge background
$E = \{E^+, E^-\}$ : Positive and negative examples
$TR$: List of derived transformation rules induced

Input: Initial set of $RD$
Output: A set of $TR$ such that $TR$ derived from $E$ and $K$
 Call the files creation procedure (specified in Algorithm 1)
//  Convert the constructs s, t $\in$ ST into a logic predicate
1:  Let $E^+ \leftarrow \emptyset$
2:  Let $B \leftarrow \emptyset$
3:  Repeat untill RT $\neq \emptyset$
4:     Convert $st_i = \{s_i, t_i\}$ into logic predicate s.t.
           $B \leftarrow B \cup s_i$  $\forall s \in S$
           $E^+ \leftarrow E^+ \cup d_i$  $\forall d \in D$
    End while
    // Generate The transformation rules TR
    TR$\leftarrow \emptyset$
    Repeat
    Create three files required by ALEPH such that:
    file.b $\subset$ B, file.f $\subset E^+$, and file.n $\subset E^-$
    ALEPH induces  a rule R = (LHS, RHS)
    TR $\leftarrow$ TR $\cup$ R
    Until $\exists$ R $\in$ TR , $\forall d_i \in E^+$
    Return TR.
```

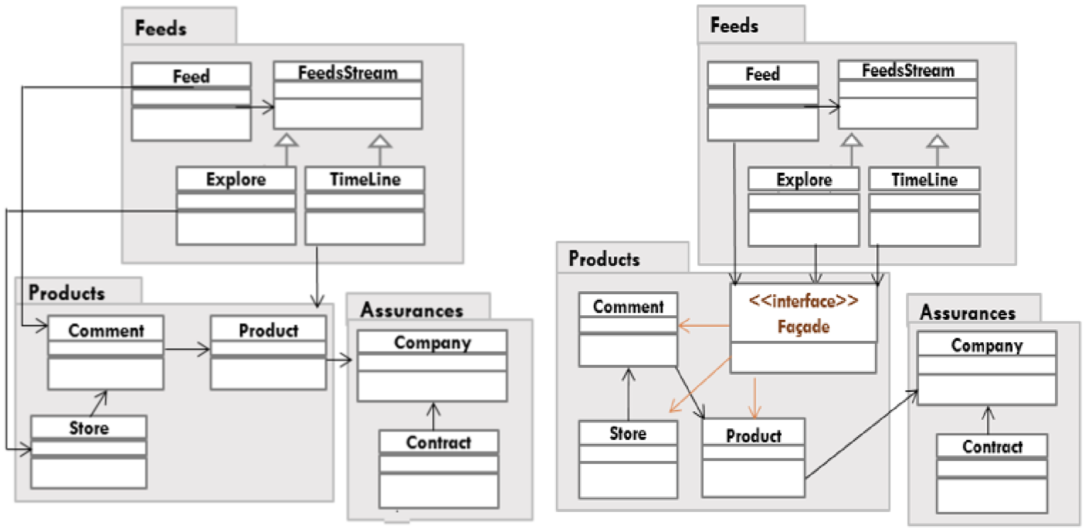
```
Algorithm 3: Transformation Rules Evaluation

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 $\leftarrow \emptyset$
2.   While TR not empty do
3.   $\forall tr_i \in$ TR translate $tr_i$ into $jr_i$
4.   JR $\leftarrow$ JR $\cup jr_i$ where $jr_i$ ={ P, C }
5.   TR $\leftarrow$ TR/ $tr_i$
6.   End while
//  Convert the given requirement models into logic predicates
7.   F $\leftarrow \emptyset$
8.   While RQ not empty do
9.   $\forall rq_i \in$RQ convert $rq_i$ into a predicate $f_i$
10.  F $\leftarrow$ F $\cup f_i$
11.  RQ$\leftarrow$ RQ/$rq_i$
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  $F \leftarrow F \cup$ fi    $\vee$    existing fact fi  is removed
16.  Convert resultant F to XMI format (SD)
```



(a) (b)

Fig. 2. The UML class diagram for analysis-design pair (a) the source model (requirement analysis) and (b) the target model (software design).



(a) (b)

Fig. 3. Part of class diagram of application (a) the source model and (b) the target model.

C. Rules Evaluation and Refinement

Before and after the rules application, different measures were used to evaluate their performance. Completeness and consistency measures refer to the positive and negative examples covered by the induced rules. This can indicate the accuracy of the induced rules based on the learning examples. Furthermore, the transformation designer can validate the correctness of the produced rules after applying them on additional test cases. The rule-created target model can be compared to the actual target for the sake of performance evaluation of the transformation rules. Several performance measures, shown in Algorithm 3, are used in this context (more details are given in Section B).

Human expert evaluation for the resulted design can be considered. This type of evaluation may help to update the priority of the rules application. In addition, expert feedback would help refining. Expert opinions may contribute to the rule base by adding new rules or relaxing the application of others. Here, we allow automatic assignment of priorities to the rules where the higher the application frequency, the higher the priority. The number of positive examples used to induce the rule determines the rule priority. That is the higher number of positive examples the higher the priority. Thus, each rule starts by initial priority equals to the number of positive examples used to induce the rule; this priority is then tuned based on the rule's application frequency and input from experts.

V. Sample Transformation Tasks

This section is dedicated to explain the two transformation problems investigated in this work. In the following, we describe each case study briefly.

A. *Packaging Class Diagram*

One of the common tasks when moving from analysis to design is the task of structuring the system classes into packages [37]. 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. In this experiment, we use our proposed approach to learn packaging rules from analysis-design pair examples. Together Fig. 2(a) and (b) represent a simple example of the analysis-design 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 (This example has been used in [28] to introduce the initial idea).

B. *Introducing Façade Design*

It is considered another high-level software design activity: introducing façade design. It is common for a class in a particular 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. For this design task, we used several examples to derive a rule for introducing façades to packages. Fig. 3 depicts one of such examples. It shows a class diagram that has many inter-packages 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. Based on the presented examples, ALEPH generalizes a hypothesis as shown in TABLE III.

VI. Experiments

This section is dedicated to the setup of the experiments performed in this work to produce a set of rules. The objective of these experiments is to provide a proof-of-concept that the proposed approach can be used to build a transformation system from requirement analysis to software design.

A. *Problem and Solution Representations*

The given UML models are presented in XMI format. To induce a general hypothesis using ALEPH, we converted the problem, represented by XMI models, to logic programs comprised two part: background knowledge and positive examples. TABLE II. demonstrates the conversion of the UML model presented in Fig. 2 and 3; with background knowledge and positive examples, respectively. In addition, the generated negative examples under CWA.

TABLE II. The Problem Representation Written in ALEPH (I.E., Prolog Syntax)

Input Type	**Packaging Class Diagram**
Types and Modes Declarations	:- modeh(*, packageOfClasses(+class, -class, -class, -class)). :- modeh(*, packageOfClasses(+class, -class, -class)). :- modeb(*, inheritance (+class, -class)). :- modeb(*, association (+class, -class)). :- modeb(*, inheritance (+class, -class)). :- modeb(*, association (+class, -class)). :- determination (package/4, inheritance/2). :- determination (package/4, association/2). :- determination (package/3, inheritance/2). :- determination (package/3, association/2).
Background Knowledge	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).
Positive Examples	packageOfClasses (book, studyBook, magazine, journal). packageOfClasses(memberRecord, faculty, student). packageOfClasses (librarian, bill, transaction).
Negative Examples	packageOfClasses (memberRecord, studyBook, bill). packageOfClasses(book, student, librarian). packageOfClasses (magazine, faculty, transaction).
Input Type	**Introducing Façade Design**
Types and Modes Declarations	:- 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).
Background Knowledge	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).
Positive Examples	packaheHasFacade(assurance, façade).
Negative Examples	packaheHasFacade(products, façade). packaheHasFacade(feeds, façade).

TABLE III. THE SOLUTION REPRESENTATION

Example	Induced Rule
Packaging Class Diagram	packageOfClasses(A,B,C) ← 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) ← 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) ← 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
Introducing Façades Example	packageHasFacade(A,B) ← 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.

In TABLE III. the second column represents samples of the rules produced by ALEPH system based on the predefined modes and given examples. In this set of rules, LHS (left-hand side) represents the conclusion (introduce a package) in order to group different classes into a single package wherever RHS (right-hand side) which is the premise is satisfied.

B. Solution Evaluation

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). The equations demonstrated in Algorithm 3 are collected during experimentation. We focused on validating the generated artifacts. To do that, we compared the generated artifacts with the actual ones provided as part of the given pair. As shown in algorithm 3, we used five different measures in the conducted experiments. Although we considered two transformation tasks, here is an explanation how we evaluate the solution in the task of packaging the classes.

TP refers to the correctly placed classes in the created package $\wp_j$ while TN refers to the classes that are not placed in $\wp_j$ correctly. FP indicates the extra classes placed in $\wp_j$ while they do not exist in p_i. Finally, FN indicates the number of classes that are exist in p_i but not placed in $\wp_j$. As a last step, we calculate the average across all packages for all the measures. It is worth mentioning that, for the experiments related to the second task, it was not applicable to calculate TN so we exclude some measures.

TABLE IV. DATASETS STATISTICS

Artifacts	Min	Max
Packages	3	27
Classes and Interfaces	10	151
Relationships	11	188

When validating the generated packages, we need to pay attention of their content. Let AD= {p_1, p_2, …, p_n} be the number of packages of the actual design. Let GD= {$\wp_1$, $\wp_2$, …, $\wp_m$} be the number of packages of the corresponding rule-created (generated) 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}, …, c_{ik} and the corresponding package p_j in GD may consists of the same, more or less classes c_{j1}, c_{j2}, …, c_{jl}.

C. ALEPH Settings

ALEPH has many settings to adjust the search process, and the ones we perform the experiments with are explained in this section. We use default ALEPH settings. Different search strategies (such as heuristic, depth first) have been used; however the results obtained were comparable. All presented results in the following came from these settings.

D. Datasets

The datasets used in the experiments comprises around 34 systems. Each system consists of the analysis and design models. These cases were collected mostly from academic projects, examples from textbooks, and by reserve engineering. Each system consists of analysis/design pair. In turn, each design system comprises at least three packages. The total number of packages in the base is 217 while the total number of classes and interfaces is 1540. TABLE IV. shows brief statistics of the systems' artifacts i.e., packages, classes, interfaces, and relationships such as association, generalization, aggregation and others.

E. Experimental Results / Quantitative Validation

This section shows the results obtained from the two conducted experiments by using the described datasets. The available examples were divided into learning set and validation set in the ratio 2:1. Each set has been selected randomly with 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 are demonstrated in TABLE V.

1) Measuring the packaging rules performance

We measured the performance of the induced rules individually. The performance of each rule was measured by applying the rule on all systems in one run. Fig. 5 shows two types of experiments that were conducted to measure the rules performance. In one experiment, shown in Fig. 5(a), individual rule performance was evaluated by applying the rule on all the learning systems, batched together. The rules showing low performance have been retracted form the rules base to avoid impact the overall system performance, e.g. rules 9-10. Then a genetic algorithm-based procedure was used to find subsets of remaining rules that gives the best results against the learning systems one by one. Fig. 5(b) presents the accuracy measures for the learning systems.

This experiment paid attention for the number of times each rule was considered to give the best accuracy with each system. Fig. 4 shows the percentage of times the rules applied. We aim here to provide a kind of score of each rule that assist selecting the rules in the future when apply the rules on real applications that have no the actual target model. To rank the rules, we need for assigning these scores (discussed in Algorithm 3). Then the rules were applied on the validation systems based on their ranking.

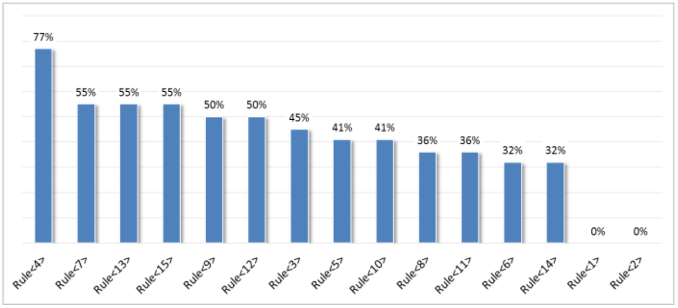
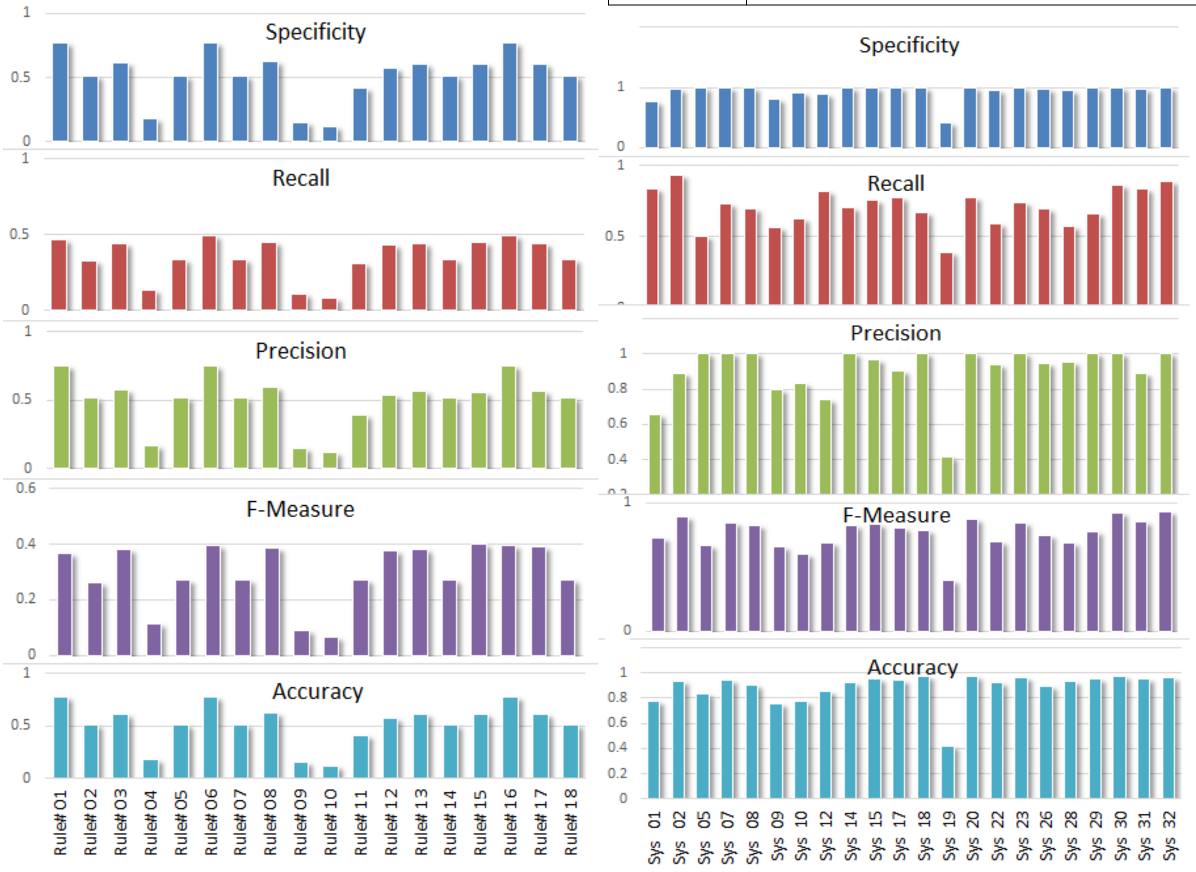


Fig. 4. Rules ranked accoding to the frquency of application.

TABLE V. SAMPLES OF THE INDUCED TRANSFORMATION RULES

Task	Induced Transformation Rule
Packaging Class Diagram	packageOfClasses(A,B)← association(A,B). packageOfClasses(A,B) ← inheritance(B,A).
	packageOfClasses(A,B,C) ← association(A,B), association(C,A). packageOfClasses(A,B,C) ← inheritance(B,A), association(A,C). packageOfClasses(A,B,C) ← inheritance(C,A), inheritance(C,B). packageOfClasses(A,B,C) ← association(B,A), association(C,A).
	packageOfClasses(A,B,C,D) ← association(A,C), association(C,B), association(A,D). packageOfClasses(A,B,C,D) ← association(A,D), association(C,A), association(C,B). packageOfClasses(A,B,C,D) ← inheritance(B,A), inheritance(B,C), inheritance(B,D). packageOfClasses(A,B,C,D) ← association(C,A), association(C,D), association(B,C).
Introducing Façades	packageHasFacade(A,B) ← packaheHasClass(C,D), associationAcrossPackages(C,D,A,E).



(a) (b)

Fig. 5. (a) Individual rule application- overall systems, (b) GA-procedure - learning systems.

2) *Validating the best induced rules*

The final rules resulted from the learning phase have been validated in this experiment against the set of validation systems. According to the rules scores, this experiment started by applying first two rules then added one rule each run. Fig. 6 demonstrates the overall average of accuracy measures resulted from validation using 12 systems with different number of rules. Obviously the performance was stable when considering 3, 4 or 5 rules, since we considered the best rules came from two-ways evaluations. Increasing or decreasing the number of rules vary form one system to another, i.e., some systems have a steady accuracy measures starting for different number of rules, while accuracy measures of others improved/impacted when adding more rules. However, these changes are slightly small, thus the overall average shows a comparable values. When applying all the rules the performance was impacted because the rules 1 and 2 were included. The two rules can group classes from different packages together. Thus we noticed that their applications frequencies equal zero when using learning samples.

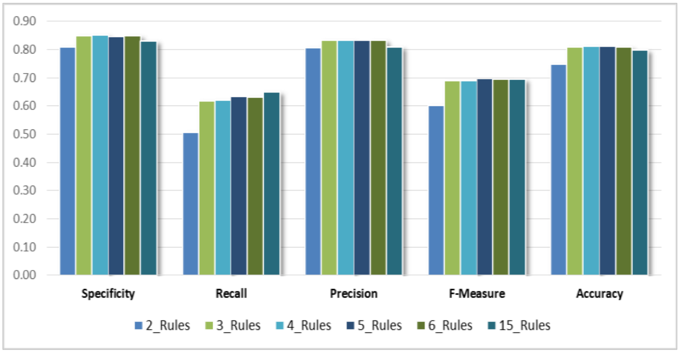
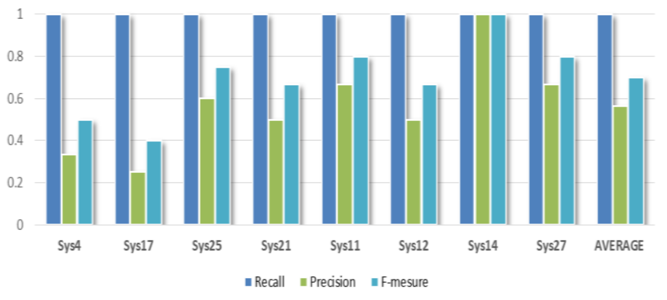
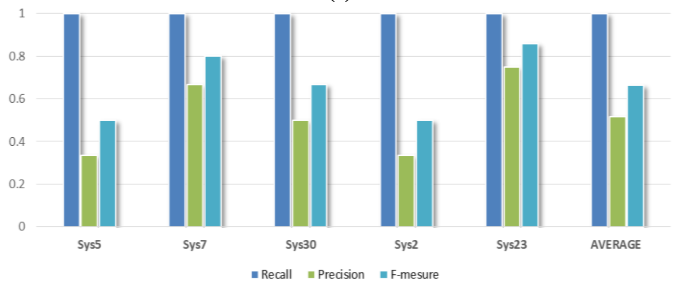


Fig. 6. Average of accuracy measures with different number of rules - validation systems.



(a)



(b)

Fig. 7. Accuracy measures of applying façade rules (a) learning results, (b) validation results.

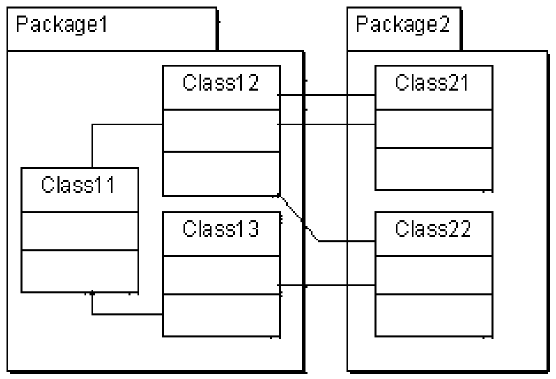
3) *Accuracy measures of Façades rules*

For introducing Façade design pattern 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, ALPEH induces only one rule for all training data. 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. Fig. 7(a) shows the accuracy measures when applying the induced unique rule on eight learning systems. It is worth mentioning that, only three measurements used for this experiment because there are true negatives can be collected here. In the same way, the induced rule has been applied on the validation systems. Fig. 7(b) demonstrates the accuracy measures when applying the rules on the validation systems.

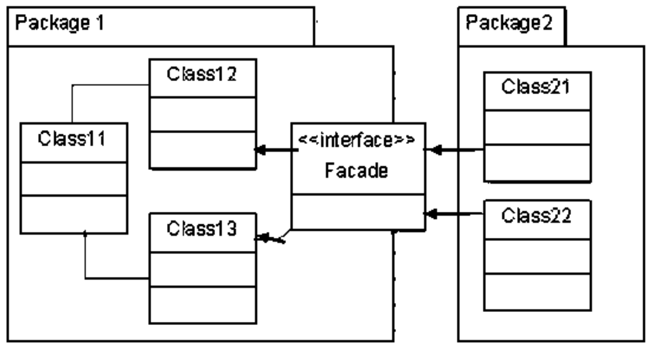
VII. DISCUSSION

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 input (-) as used in TABLE II. 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. Owing to the space limitation of this paper, we ignore these details. 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 are seen in the given examples while training, it will not be possible to synthesize the right output. In reality, for one example has 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 consist 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 Section E vary from one system to another. The reason behind that is the nature of the used examples to generate the transformation rules. 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. In the other hand, the relations among the classes are not easy to be covered by the already induced rules. The application of the rules follows their ranking which shows their how many times they were selected to get the best accuracy. When applying the rules against the validation set, small set of rules can give a comparable accuracy measures. When adding more rules means that, more classes can be grouped in incorrect packages.



(a)



(b)

Fig. 8. Example to show the drawback of the derived rule.

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 Fig. 6. The reason behind that is that the rule will introduce an interface between two packages whenever there is a relation between their classes.

A. Threat to Validity

The main threats to validity, as with any software engineering research, are the data scarcity 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 different resource have been considered to collect the datasets (student projects, textbooks, reverse engineering). Using different sources helps ensuring that the datasets are collected in unbiased manner. In addition, selecting randomly learning systems that are different from the validation set 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. Another threat to validity of this work is the incompleteness in terms of transformation problems coverage. We have considered two major designs activities to show the power of ILP in generalizing rules in MDD context. Future effort will try to make contact with some potential 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.

B. Open Issues and Future Work

In this section, we discuss some open issues related to the usage of ALEPH system to derive analysis-design transformation rules. Almost all the current ILP systems, including ALEPH, enforce modes declarations for any clause hypothesized by the ILP system. That is, it is supposed to predefine the head and body of the target hypothesis.

ALEPH system uses the given background knowledge along with the given examples to 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. In the conducted experiments, many examples have not been covered using ALEPH because the relations represented are unique i.e. no similar example especially for the packages have many classes.

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 artifacts surely influences corresponding design decisions. A simple example is shown in Fig. 8 to give a glance of this shortcoming. Fig. 8(a) (source model) depicts that there are four relations linking *"Package1"* to *"Package 2"*. Based on these relations, a façade is introduced shown in Fig. 8(b) (target model). 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 façades. For further discussion of the recorded limitations in this context, the reader may consultant our recent work [38]. In future work, there is a need to investigate more the aforementioned open issues and to find appropriate solutions.

VIII. Conclusion

Different model transformation by examples (MTBE) approaches have been proposed in the literature. However, none of the proposed approaches tried to tackle the analysis-design transformations problem using ILP. 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.

We conducted experiments using 34 systems with different sizes and form different sources. The systems were divided into learning and validation sets. The obtained performance measures show that the approach is promising. The more examples presented to the system, the more trustful rules the system can generate.

ACKNOWLEDGMENT

The authors would like to acknowledge the support provided by King Abdulaziz City for Science and Technology (KACST) through the Science & Technology Unit at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No.11-INF1633-04. as part of the National Science, Technology and Innovation Plan.

REFERENCES

[1] Varró, "Model Transformation by Example," in In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, 2006, pp. 410-424.

[2] X. Dolques, et al., "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.

[3] D. Zhang and J. J. P. Tsai, Machine Learning Applications in Software Engineering: World Scientific Inc., 2005.

[4] H. A. Al-Jamimi and M. Ahmed, "Machine Learning-based Software Quality Prediction Models: State of the Art," presented at the The 4th International Conference on Information Science and Applications, June 24-26, Pattaya, Thailand, 2013

[5] 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.

[6] H. A. Al-Jamimi and M. Ahmed, "Prediction of Software Maintainability Using Fuzzy Logic," in 3th IEEE International Conference on Software Engineering and Service Science (ICSESS 2012), Beijing, China, 2012.

[7] H. A. Al-Jamimi and L. Ghouti, "Efficient prediction of software fault proneness modules using support vector machines and probabilistic neural networks," in 5th Malaysian Conference in Software Engineering (MySEC), 2011.

[8] H. A. Al-Jamimi, "Toward comprehensible software defect prediction models using fuzzy logic," in Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on, 2016, pp. 127-130.

[9] S. Muggleton and L. D. Raedt., "Inductive logic programming: Theory and methods," The Journal of Logic Programming, vol. 19 pp. 629-679, 1994.

[10] S. Muggleton, "Inductive logic programming," New Generation Computing, vol. 8, pp. 295-317, 1991.

[11] F. A. Lisi, "Learning Onto-Relational Rules with Inductive Logic Programming," in Proceedings of CoRR, 2012.

[12] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," IBM Systems Journal, vol. 45, pp. 621-645, 2006.

[13] M. Faunes, et al., "Genetic-programming approach to learn model transformation rules from examples," presented at the Theory and Practice of Model Transformations, 2013.

[14] 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.

[15] T. M. Mitchell, "Generalization as search," Artificial intelligence vol. 18, pp. 203-226, 1982.

[16] J. R. Quinlan, "Learning logical definitions from relations," Machine Learning, vol. 5, pp. 239-266, 1990.

[17] S. Muggleton and C. Feng, "Efficient induction of logic programs," Inductive logic programming, vol. 38 pp. 281-298, 1992.

[18] S. Muggleton, "Inverse entailment and Progol," New generation computing, vol. 13, pp. 245-286, 1995.

[19] A. Srinivasan, "The Aleph Manual," University of Oxford, 2007.

[20] W.-J. Hou and H.-Y. Chen, "Rule Extraction in Gene-Disease Relationship Discovery," Gene vol. 518, pp. 132-138, 2013.

[21] Z. Balogh and D. Varró, "Model transformation by example using inductive logic programming," Software and Systems Modeling, vol. 8, pp. 347-364, 2009.

[22] P. Ferreira, et al., "Interpretable models to predict Breast Cancer," in Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on, 2016, pp. 1507-1511.

[23] A. Srinivasan and M. Bain, "An empirical study of on-line models for relational data streams," Machine Learning, vol. 106, pp. 243-276, 2017.

[24] J. Van Haaren, et al., "Analyzing volleyball match data from the 2014 World Championships using machine learning techniques," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 627-634.

[25] V. Vercruyssen, et al., "Qualitative spatial reasoning for soccer pass prediction," 2016.

[26] M. Wimmer, et al., "Towards Model Transformation Generation By-Example," in In: 40th Hawaiian Int. Conf. on Systems Science (HICSS 2007), 2007.

[27] F. Jouault, et al., "ATL: a model transformation tool," Science of Computer Programming, vol. 72 pp. 31-39, 2008.

[28] X. Dolques, et al., "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.

[29] M. Huchard, et al., "Relational concept discovery in structured datasets," Annals of Mathematics and Artificial Intelligence, vol. 49, pp. 39-76, 2007.

[30] H. Saada, et al., "Generation of Operational Transformation Rules from Examples of Model Transformations," presented at the Model Driven Engineering Languages and Systems, 2012.

[31] I. García-Magariño, et al., "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.

[32] M. Faunes, et al., "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.

[33] S. Muggleton, et al., "ILP turns 20," Machine Learning, vol. 86, pp. 3-23, 2012.

[34] S. Sankaranarayanan, et al., "Mining library specifications using inductive logic programming," in ACM/IEEE 30th International Conference on Software Engineering, 2008.(ICSE'08). , Leipzig, Germany, 2008.

[35] H. A. Al-Jamimi and M. A. Ahmed, "Knowledge acquisition in model driven development transformations: An inductive logic programming approach," in TENCON 2014- 2014 IEEE Region 10 Conference, 2014, pp. 1-6.

[36] M. d. C. Nicoletti, et al., "Learning temporal interval relations using inductive logic programming," presented at the In Integrated Computing Technology, 2011.

[37] M. O'Docherty, Object-Oriented Analysis and Design Understanding System Development with UML 2.0: John Wiley & Sons Ltd, 2005.

[38] H. A. Al-Jamimi and M. A. Ahmed, "Learning Requirements Analysis to Software Design Transformation Rules By Examples: Limitations of the Current ILP systems.," in 5th IEEE International Conference on Software Engineering and Service Science (ICSESS 2014), Beijing,China, 2014