FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

TRANSFORMING USER REQUIREMENTS TO TEST CASES USING MODEL-DRIVEN SOFTWARE ENGINEERING AND NATURAL LANGUAGE PROCESSING

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Sai Chaithra Allala

2023

To: Dean John L.Volakis
College of Engineering and Computing

This dissertation, written by Sai Chaithra Allala, and entitled Transforming User Requirements to Test Cases using Model-Driven Software Engineering and Natural Language Processing, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Xudong He

Masoud Sadjadi

Monique Ross

Debra Vandermeer

Tariq M. King

Peter J. Clarke, Major Professor

Date of Defense: March 23, 2023

The dissertation of Sai Chaithra Allala is approved.

Dean John L.Volakis
College of Engineering and Computing

Andr´es G.Gil
Vice President for Research and Economic Development and
Dean of the University Graduate School

Florida International University, 2023

DEDICATION

I am very grateful to my past and present research fellow-mates, especially Juan P. Sotomayor, Dionny Santiago, and Hamilton Chevez. They always motivated me, and it has been a pleasure to work with them. Finally, I take this opportunity to dedicate this dissertation to my mother Arundhathi Adumala, my father Laxma Reddy Allala, my sister Prathyusha Allala, my brother-in-law Krishna Bunny Nagavolu and her family. Although we have been thousands of miles apart, they always had me in their prayers and wishes. Also, I would like to thank my friends that I made along the way, specially Keerthi Tirumalaraju. Your love and encouragement have been my driving force, and I am forever grateful.

This dissertation would not have been possible without these individuals' support and encouragement, and I am deeply thankful for their contributions.

ACKNOWLEDGMENTS

First and foremost, I sincerely thank my advisor, Dr. Peter J. Clarke, for always guiding, motivating, sharing ideas, and encouraging me throughout my Ph.D. program. I want to express my gratitude to the committee members Dr.Xudong He, Dr.Masoud Sadjadi, Dr.Monique Ross, Dr.Debra Vandermeer, and Dr.Tariq M. King for their unreserved help and support. I want to take this opportunity to acknowledge the various facilities provided by the Knight Foundation School of Computing and Information Sciences that have aided in completing this dissertation. I want to extend my gratitude to all the funding agencies that enabled the completion of this dissertation.

ABSTRACT OF THE DISSERTATION

TRANSFORMING USER REQUIREMENTS TO TEST CASES USING MODEL-DRIVEN SOFTWARE ENGINEERING AND NATURAL LANGUAGE PROCESSING

by

Sai Chaithra Allala

Florida International University, 2023

Miami, Florida

Professor Peter J. Clarke, Major Professor

Testing continues to be the main approach to ensuring software quality during development. Although there have been many attempts to automate the generation of test cases from user requirements (formal or informal), creating test cases continues to be mainly a manual process. However, many studies have shown that automating the generation of test cases from requirements can substantially reduce costs and improve the efficiency of the testing process. Test automation has also proven to show positive effects on software quality. With the advances in Model-driven Software Engineering (MDSE), Artificial Intelligence (AI), and Natural Language Processing (NLP), the possibility of further automating the generation of test cases from requirements is increasing.

We present an automated test case generation approach that uses a model-to-model (M2M) transformation that converts user requirements into test cases with the support of a knowledge base and NLP. The key to this transformation process is defining meta-models for user requirements (use cases and user stories) and test cases. The proposed M2M transformation is composed of three phases. Phase 1 includes creating Enhanced User Requirement Models (EURMs) from user requirements using NLP analysis and an application-specific data model. Phase 2 involves transforming EURMs to Abstract test Cases (ATCs) using the transformation rules defined between

EUR and user requirements meta-models. Phase 3 consists of generating concrete test cases from ATCs and an application-specific data model instance. To validate each phase of the transformation process, case studies were conducted using a tool developed to evaluate the feasibility and accuracy of converting user requirements to test cases.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ATC	Abstract Test Case
AI	Artificial Intelligence
ATL	Atlas Transformation Language
EMF	Eclipse Modeling Framework
EUR	Enhanced User Requirements
EURM	Enhanced User Requirements Model
FIU	Florida International University
FN	False Negatives
IE	Information Extraction
MDSE	Model Driven Software Engineering
M2M	Model to Model
NLP	Natural Language processing
OCL	Object Constrained Language
PMS	Payroll Management System
TC	Test Case
TCP	True Positives
UC	Use Case
UML	Unified Modeling Language
US	User Story
UR	User Requirements
URM	User Requirements Model

CHAPTER 1

INTRODUCTION

The software process usually consists of a number of phases such as requirements, design, development, testing and maintenance. An important aspect of the software process is the consistency of the artifacts developed starting from the requirements through software design and implementation [27].

Requirements, written as either user stories or use cases, are the basis for developing system test cases and help the developer to answer the question are we building the right product. The testing of software is becoming increasingly difficult as the complexity of systems continues to grow. It is now frequently the case that more than 50% of development time may be devoted to testing [39]. The main objective of testing a system is to check if the system aligns to its requirements.

During the past decade, there have been many attempts to automate the generation of test cases from user requirements (formal or informal) [23, 32, 35, 50, 52] with some success. Although there has been some progress in automating the generation of test cases from requirements, it is still mainly a manual process specifically, the oracle problem [7]. Several approaches have been presented in the literature that automatically generates test cases from use case specifications however, these techniques tend to be cumbersome and require many resources [53]. Many studies have shown that automating the generation of test cases from requirements can substantially reduce costs and improve the efficiency of the testing process [36]. However, more needs to be done to keep pace with the advances in other areas of software development, and some of these advances can even be integrated into the software testing process. Three such areas with major advancements include Model-driven Software Engineering (MDSE), Artificial Intelligence (AI), and Natural Language Processing

(NLP). Combining different techniques from these areas has the potential to significantly automate the generation of test cases from requirements, thereby reducing the overall cost of testing and improving the efficiency of the testing process. There are several tools and techniques used in MDSE that may help to alleviate the problem of automatically generating test cases from use cases or user stories with a reduced overhead.

MDSE has emerged as a popular and commonly used method for designing software systems in which models are the primary development artifact. MDSE has resulted in the trend toward further automating the software process. Given that most user requirements are written in natural language, the recent advances in NLP and MDSE provide an opportunity to further automate the test generation process. One of the major problems that NLP aims to solve is the high learning curve for most of the automation tools that need some programming language to be learned to automate test cases [8]. NLP supports the creation of test cases using Natural Language [1], which means there is no specific set of rules that needs to be learned or understood.

Our approach to generating test cases from user requirements (use cases or user stories) is based on a model-to-model transformation to convert requirements into test cases. Key to this transformation process is the use of meta-modeling for requirements and test cases.

The proposed M2M transformation is composed of three phases. Phase 1 includes creating Enhanced User Requirement Models (EURMs) from user requirements using NLP analysis and an application-specific data model. Phase 2 involves transforming EURMs to Abstract test Cases (ATCs) using the transformation rules defined between EUR and user requirements meta-models. Phase 3 consists of generating concrete test cases from ATCs and an application-specific data model instance. To validate

each phase of the transformation process, case studies were conducted using a tool developed to evaluate the feasibility and accuracy of converting user requirements to test cases.

1.1 Problem Definition

The problem statement to be investigated is ***"How can the process of translating natural language requirements to test cases be further automated, given the recent advances in MDE and NLP?."*** To answer this question, the research effort required a division into the following three sub-problems. Each sub-problem noted below is referred to as SP-I, SP-II, and SP-III further in this dissertation, respectively. The problem is sub-divided as follows:

1. ***SP-I:*** How can user requirements written as structured Natural Language along with the associated data model be represented in a semi-formal manner to support testing?

2. ***SP-II:*** How can abstract test cases be generated given the semi-formal representation of user requirements model?

3. ***SP-III:*** How can concrete test cases be generated given abstract test cases and an instance of a data model derived from user requirements?

The three Sub-Problems mentioned above are the steps required to attain the main objective. They are concrete and specific. Our SP-I helps us standardize the input user requirements. Our SP-II establishes the rules between the input(requirements) and output(test cases). Finally, SP-III queries the data model instance to find relevant test data. Ultimately they all provide approach paths that lead to the overall problem.

1.2 Overview of Solutions

In Chapter 2 a comprehensive and critical examination of the existing body of knowledge relevant to the topic being studied. A literature review is presented that contains background material that includes definition of terms and explanation of key concepts used throughout the dissertation. Background is followed by related work of the research problem and the methods used it includes an overview of key theories, concepts, and previous studies, and identifies gaps and opportunities for further research.

Chapter 3 describes a high-level overview of the research problem and the methods used to provide a clear and comprehensive summary. It provides a bird's-eye view of the entire dissertation with an overview of the structure and organization brief introduction to the key chapters and sections. This high level Overview shows how the three sub -problems are integrated to form a total solution. It outlines the methods and techniques used to briefly and concisely answer the research question.

Chapter 4 addresses the sub-problem SP-I that the dissertation aims to answer. It includes a description of the design, detailed explanation of the meta-model for User Requirements (URs) from both Use Cases and User Stories. Followed by experiment setup and analysis methods for validation of the design. The results of the data analysis is presented, and the findings are discussed in relation to the sub problem. The EUR model is created from the user requirements model (use cases) and a data model (ER diagram) using NLP techniques. An algorithm is developed to generate a EUR model, which is further validated to our EUR meta-model. To show the feasibility of approach we developed a tool that can accept a cross-section of requirements using 50 use cases and 50 user stories.

Chapter 5 addresses the sub-problem SP-II. An ATC meta-model and an EUR meta-model is built to accommodate the transformations. Define rulers and helper functions to perform transformations from EUR to ATC meta-models using ATL.

Model-to-Model Transformations from EUR meta-model to Abstract Test Case (ATC) meta-model generates ATC models. Comparing manually generated ATC's to generated ATC's provides the validity and reliability of the approach. To show the feasibility of the approach, we extended the tool and conducted a case study with use cases and abstract test cases from a Payroll Management System.

Chapter 6 addresses the sub-problem SP-III. UR2TC-GenTool connects the generated ATC model to the representing the data model used in the implementation of the PMS application and the associated data model instances. Execute appropriate queries to built both sunny and rainy day test cases for two use cases.

Chapter 7 summarizes the main findings of the study and their implications for theory, practice. It also provides recommendations for future research and identifies the limitations of the study and its findings. There are seven appendices; Appendix A contains an Example for User requirement used repeatedly in the dissertation. . Appendix B contains algorithms referenced in the Chapter 4. Appendix C shows XMI Representations for the EUR-MM and a Model Instance. Appendix D shows XMI Representations for the ATC-MM. Appendix E shows ATL Transformation Rules. Appendix F displays Abstract Test Case Model generated, and the last appendix contains bio and the publications.

1.3 Contributions

The contributions of the work presented in the dissertation include the following:

- The refinement and development of meta-models to support creating test cases from user requirements. A meta-model for user requirements(UR) specifications that support user stories and use cases was refined based on descriptions of user requirements found in the literature, see Section 4.1. A meta-model for enhanced user requirements (EURs) was developed that extends the UR meta-

model to include features of the data model used in a domain application, see Section 4.4.1. An ATC meta-model was developed to support the transformation from EUR models to abstract test cases (ATCs), see Section 5.1.1.

- The creation of algorithms to support the pre-processing of user requirements using NLP and the transformation from EUR models to ATCs. Algorithm 4.1 performs pre-processing of user requirements by using NLP to extract structured features that support the creation of test cases, see Section 4.2.2. Algorithm converts a UR model and an applications data model into an EUR model. This conversion is part of the pre-processing needed to transform EUR models into ATCs, see Section 4.4.2.

- A transformation process that generates ATCs from EUR models. The transformation process includes defining rules and helper functions between the EUR and ATC meta-models, see Section 5.1.2.

- A tool that takes as input a set of user requirements, a data model, a data model instance, and generates a set of concrete test cases. This overarching tool is divided into smaller component tools, each allowing for the validation of a major component of the test case generation process from URs to concrete test cases. Sections 4.2, 5.2, and 6.1 provide descriptions of the smaller component tools and the overall tool, respectively.

- Case studies were performed that validated different parts of the overall process to convert user requirements to concrete test cases. The first case study was used to validate the UR meta-model using requirements from academia and industry, see Section 4.3. The second case study was used to validate the creation of EUR models and the transformation of EUR models to ATCs, see Section 5.3. The

last case study shows how the overall process can be automated to generate concrete test cases from user requirements, see Section 6.2.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we provide the background essential to understanding the problem under investigation and review prior works related to user requirements, testing, modeling, NLP and the transformation process. The related work section provides a comprehensive and critical examination of the relevant literature. It provides an overview of existing studies, theories, and methods that are relevant for automatically generating test cases using MDE and NLP. The purpose of this chapter is to demonstrate an understanding of the existing body of knowledge, to identify gaps and opportunities with existing research.

2.1 Background

In this section, we provide a brief overview of user requirements, test cases, metamodels, model transformations and natural language processing.

2.1.1 Software User Requirements

Users generally communicate the needs of a software application by stating the requirements in a natural language. These requirements are therefore stated informally and as a result are inherently ambiguous, thereby making the requirements somewhat difficult to understand, interpret, model and maintain. To assist with improving the understandability of the requirements software modeling languages are used to create semi-formal models of the requirements. One such language that has gained widespread use in industry is UML (Unified Modeling Language) [49]. The two most common techniques for requirements elicitation are use cases and user stories due to their prominent comprehensibility, decomposability and interactivity [54].

Table 2.1: Use case template for Functional requirements [14].

USE CASE #	<the name is the goal as a short active verb phrase>
Goal in Context	<a longer statement of the goal, if needed>
Preconditions	<what we expect is already the state of the world>
Success End Condition	<the state of the world upon successful completion>
Failed End Condition	<the state of the world if goal abandoned>
Trigger	<the action upon the system that starts the use case>
Description	<Action-put here the steps of the scenario from the trigger to goal delivery,and any cleanup after>
Extensions	<Branching Action-condition causing branching>
Sub-Variations	<Branching Action-list of variations>

A *use case* represents a piece of functionality in the system that provides the user with results or value [31]. Use cases therefore, capture what the customers and users expect the system to do after it is developed. In addition, they reveal process alternatives, process exceptions, undefined terms, and outstanding issues. Test cases are usually generated from scenarios or instances of use cases [13].

A *user story* is the smallest unit of work in an agile framework [15]. It's an end goal, not a feature, expressed from the software user's perspective. User stories are a few sentences in simple natural language that outline the desired outcome and don't go into much detail. Requirements are added later, once agreed upon by the team. The use cases and user stories have different, although overlapping, goals, and different means of achieving those goals. Furthermore, they can co-exist and support each other [16].

As a <Role>, I want <feature> so that <execution of feature or post condition>.

Table 2.2: Use case template for non-Functional requirements [10].

Usability	how operable, training and communication
Reliability	how accurate is the software
Performance	how speed efficiency and response time
Supportability	how many configurations and compatibility
Implementation	which languages are associated languages

Table 2.3: Use case template for related information [14]

Risk	the level of importance
Frequency	how often is it expected to happen
Priority	how critical to your system
Related Use cases	all the related use cases

Table 2.4: User story template and illustrative example.

Template		Example
As a	*<type of user>*	As a company,
I want	*<some goal >*	I want my Oregon employees to sign the mandatory direct deposit disclosure
so that	*<some reason >*	so that we remain in compliance

Table 2.4 shows the template used for writing user stories, along with an example.

2.1.2 Test cases

In the simplest definition, a *test case* is a set of conditions or variables under which a tester determines whether the software satisfies requirements and functions properly [47]. You can think of a test case as a set of step-by-step instructions used to verify that something behaves as it is required to behave. The objective of testing is to

verify and validate requirements by defining a test set (a number of test cases) and designing a rationale. During this process, inspection and review techniques are vitally important to verify requirements and to identify testable requirements [47]

We define the structure of a test case as having five (5) components. These include (1) *Test Case ID* - unique identifier for a test case; (2) *Purpose* - a unique reason stating the objective of the test case; (3) *Test Setup* or *Precondition* - the state the entity under test needs to be in before the test case should be executed; (4) *Input* - the values that should be entered by the tester; and (5) *Expected Output* - how the system reacts to the input values, this may include a user interface to update or change of state in the system.

2.1.3 Meta-models

A *meta-model* in simple terms is defined as a model of a model. Atkinson and Kühne [9] state that this definition may be an over simplification of the concept of a meta-model. A *meta-model* can be defined as an abstraction of a model highlighting the properties of the model itself [11]. If a model adheres to the properties of the meta-model, then the model conforms to the meta-model. The concept of a meta-model should be viewed from two separate and orthogonal dimensions, one from the view of a language definition and the other from the view of a domain definition. These two dimensions result in linguistic meta-modeling and ontological meta-modeling [9].

Atkinson and Kühne [9] describes an example taken from the domain of dogs that illustrates these differences. A dog *Breed* is a linguistic instance of a *Metaclass*, a *Collie* is a linguistic instance of a *Class*, while a *Collie* is an ontological instance of a *Breed*, and a *Class* is an ontological instance of a *Metaclass*.

2.1.4 Model Transformations

Model transformation includes a set of rules that define how one model (M_2) can be created from another model (M_1). *Model-Driven Engineering (MDE)*, as described by Brambilla et al. [11], includes all model-based tasks involved in the complete software engineering process, e.g., model-based evolution and model-based testing [11]. The key concepts of MDE used are meta-models and model transformations. The rules defined by the transformation are at the level the meta-models, where MM_1 is the meta-model for M_1 (source model) and MM_2 is the meta-model for M_2 (target model) [11, 42] shown in Figure 2.1. We refer to this type of transformation as an $M2M$ transformation. Since our $M2M$ transformation will use two different meta-models and the output model will be generated from scratch, we use an *exogenous out-place* transformation. We use the ATLAS Transformation Language (ATL) [33] to perform our exogenous out-place $M2M$ transformation. ATL is a hybrid transformation language that contains a mixture of declarative and imperative constructs. ATL performs only unidirectional transformations, operating on read-only source models and producing write-only target models [33].

2.1.5 Natural Language Processing

Natural Language Processing (NLP), a component of Artificial Intelligence and Computational Linguistics [23], has the goal of performing human-like language processing [38], usually with the assistance of a computer program or some other technology. NLP requires several levels of language processing that may interact with each other [38]. These levels usually include the following [23, 38]. (1) *Morphology* - parts-of-speech (POS) tagging - categorizing nouns, verbs, adverbs, etc.; stemming - reducing derived words to their base; name-entity recognition (NER) - allocating semantics to words in the text. (2) *Syntax* - constituency/dependency parsing to uncover the gram-

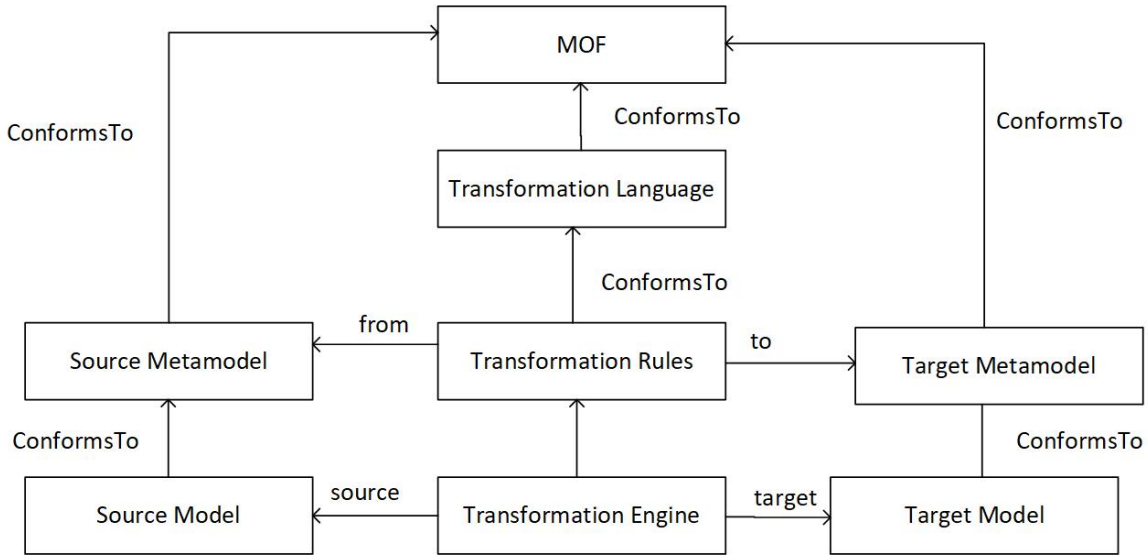

Figure 2.1: Basic Model to model transformation [18]

matical structure of the sentence (e.g., subject-verb-object structure). (3) *Semantic* - semantic role labeling, co-reference resolution, and word-sense disambiguation in a sentence. (4) *Discourse* - analyzes the text as a whole by making connections between sentences. (5) *Pragmatic* - used to determine the extra meaning read into text without actually being encoded in the text; this requires world knowledge, including intentions, plans, and goals.

There are also different categories of processing in NLP [38]. These categories include: symbolic - performs a deep analysis of linguistic phenomena; statistical - approximate linguistic phenomena using large corpora of text; connectionist - combines statistical learning with various theories of representation; and hybrid.

One of the most widely used tools for NLP is Stanford CoreNLP [41], which is a toolkit that provides an extensible pipeline that provides core natural language analysis. The Stanford CoreNLP consists of several components that perform the tasks described in the previous paragraph. These components include: Tokenization - splits text into a sequence of tokens; Sentence Splitting - splits a sequence of tokens into

sentences; POS tagging - labels tokens with their POS tags in Figure 2.5; Syntactic Parsing - provides a complete syntactic analysis based on a probabilistic parser; Sentiment Analysis - uses deep learning to annotate the parse trees; and Conference Resolution - creates a conference graph that allows for various annotations.

2.2 Related Work

The research literature contains a broad spectrum of work on automatically generating test cases using MDSE [26], and more recently using NLP [23]. Most of the related work use the term Model-Driven Engineering (MDE) to refer to MDSE. However, few works use both MDSE and NLP to generate test cases automatically. We first review the work related to MDSE, then those related to NLP, and finally those with both MDSE and NLP.

2.2.1 Model-Driven Engineering

MDE: Jin et al. [32] present a systematic literature review (SLR) on the generation of test cases from UML diagrams. Their analysis focused mainly on the model type to generate test cases, the intermediate formats used in the transformation process, and the coverage criteria used for generating the test cases. Six research questions were investigated that targeted the focus of the SLR, and based on their selection criteria, 62 primary studies were selected from 443 identified. The results of the SLR showed that the state diagram was the most popular model used (22 studies), followed by sequence diagrams (18), then activity diagrams (17). The least used diagram was object diagrams (2). This result is expected since the dynamic diagrams that are more formal are expected to be used more frequently, as they provide the necessary information needed to generate test cases. Most of the intermediate formats used in the studies were tree graphs using some tree traversal algorithm. The strongest

Table 2.5: Penn Treebank description of the POS Tags

Tag	**Description**
CC	conjunction, coordinating
CD	cardinal number
DT	determiner
EX	existential there
FW	foreign word
IN	conjunction, subordinating or preposition
JJ	adjective
JJR	adjective, comparative
JJS	adjective, superlative
LS	list item marker
MD	verb, modal auxiliary
NN	noun, singular or mass
NNS	noun, plural
NNP	noun, proper singular
NNPS	noun, proper plural
PDT	predeterminer
POS	possessive ending
PRP	pronoun, personal
PRPS	pronoun possessive
RB	adverb
RBR	adverb, comparative
RBS	adverb, superlative
RP	adverb, particle
SYM	symbol
TO	infinitival to
UH	interjection
VB	verb, base form
VBZ	verb, third person singular present
VBP	verb, non-third person singular present
VBD	verb, past tense
VBN	verb, past participle
VBG	verb, gerund or present participle
WDT	*wh*-determiner
WP	*wh*-pronoun, personal
WPS	*wh*-pronoun, possessive
WRB	*wh*-adverb

coverage criteria used were all paths, or some variation, due to the large number of possible paths in the tree/graph. Another powerful coverage criteria used was

MC/DC coverage which focuses on the combinations of conditions within decisions. In the subsequent paragraphs in the MDE section, we focus on specific approaches used.

Lamancha et al. [37] describe a model-driven framework that generates an oracle automatically by performing M2M and model-to-Text (M2T) transformations using a UML meta-model and QVT and MOF2Text transformation languages. The input consists of design models - UML sequence, class, and state machine diagrams. The output is test code consisting of parametrized input test data (XUnit code), test case procedure (test architecture code), and the automated generation of the expected output (oracle code). The M2M transformations use the UML 2.0 Testing Profile.

Gutiérrez et al. [26] present an approach to generating functional test cases from function requirements. The approach uses four meta-models and four transformations. The first transformation occurs between Functional Requirement meta-models, an endogenous transformation within the same meta-model. The second and third transformations are from the Functional Requirement meta-model to the Test Scenarios and Test Values meta-models. The final transformation occurs from the three previously used meta-models to the Test Case meta-model. The Test Scenario meta-model includes concepts that support test coverage criteria on the specification, e.g., path analysis. The Test Values meta-model includes concepts for test value generation, such as the category partition method. The transformations are implemented using QVT and the Java language.

Hue et al. [28] proposes an MDE approach (USLTG) that takes as input a use case model and a class diagram and transforms it into a model in the Test Case Specification Language (TCSL). The authors developed TCSL as part of their work. USLTG uses three algorithms to transform a use case model into a TCSL model. The algorithms in USLTG are used in a pipeline manner to (1) generate use case scenarios

and constraints, (2) generate test input data for each scenario, and (3) generate a TCSL model. The use case model supports selecting and using test coverage criteria with a UML activity diagram, thereby supporting path coverage. The paper describes the USLTG implementation and a case study showing how test cases are generated for a library application.

Our approach does not use any static or dynamic UML models that represent the system requirements. However, we use many of the transformation and meta-model concepts key to MDE. We generate a user requirements instance model directly from the natural language user requirements and the data model, then generate abstract test cases using M2M transformations. Our approach is similar to Figure 3 in Jin et al. [32], The Common Process for Model-based Testing (MBT), except for the test criteria used as input.

2.2.2 Natural Language Processing

NLP: Garousi et al. [23] conducted a systematic literature mapping (SLM) on the approaches used to extract test cases from natural-language requirements using NLP. The authors of this SLM included 67 academic peer-reviewed papers and 38 associated tools from 2001 to 2017. The SLM focused on three broad categories of research questions, including NLP-assisted software testing - which refers to any NLP-based techniques or tools that assist any software testing activity, and those works that include empirical case studies. The SLM revealed that NLP approaches used in software testing include morphology, syntax, and semantics, among others. As previously stated, the most common tool used found in the SLM was the Stanford CoreNLP [41]. The natural language requirements used both a restricted and non-restricted format. One of the main issues raised by the authors was the accuracy score of the NLP-based test generation approaches, currently between 70% and 90%.

Thummalapenta et al. [51] present a technique that automates test generation starting with a test case written in natural language and generates a sequence of procedure calls with parameters that can be automatically executed. The test case in natural language uses a restricted vocabulary that makes POS tagging easy using a natural language parser. The first phase is to break a test step (two or more segments) into preliminary segments based on conjunction words. The second phase is to tag each preliminary segment, identifying the noun (subject), verb, a second noun (object). The third phase removes ambiguities from the preliminary segments. This is done by modeling the segments as a non-deterministic program by building a test-flow graph and using backtracking to find the correct deterministic path. Human input is sometimes needed to generate the deterministic path. Three optimizations are used to reduce the search space of the test-flow graph. These include look-ahead static checking, local backtracking, and active learning. The authors report that (1) using their approach on a suite of 33 end-to-end manual tests, they were able to automate 82% without human intervention, and (2) their tool was able to reconstruct test cases from a large corpus with 88% precision and 98% recall.

Carvalho et al. [12] present NAT2TEST$_{SCR}$, a model-based testing (MBT) strategy for generating test cases from natural language requirements that uses Software Cost Reduction (SCR) as an intermediate and hidden formalism. The requirements are written using the grammar for a controlled natural language (SysReq-CNL) to minimize textual ambiguity. NAT2TEST$_{SCR}$ strategy involves three phases, the syntactic analysis - which includes NLP morphological and syntactic analyses, semantic analysis - maps the syntax trees into the semantic representation, and the SCR generation - delivers an SCR specification to help with test case generation. The strategy was evaluated using examples from four different domains vending machine, simplified control system (nuclear power), priority command function (aviation), and turn indi-

cator (automotive). To determine the effectiveness of the strategy mutation testing was used, and the mutation scores were just as good as the random test case generator Randoop [45] for the applications. The strategy outperformed the test cases written by manual testers by killing more mutants for the aviation application.

Fischbach et al. [22] describe an approach to generate test cases automatically from user stories by applying NLP to the acceptance criteria in the user stories. The approach, referred to as the SPECMATE pipeline, takes as input user stories' acceptance criteria, performs dependency parsing to generate a dependency tree, traverses the tree to generate a cause-effect-graph (CEG), and finally generates a minimal set of test cases from the CEG using the Basic Path Sensitization Technique (BPST). The authors initially analyzed 961 user stories from an industry application and determined that between 31% to 51% contained acceptance criteria that exhibited recurring formulation patterns (cause-effect patterns). The approach was evaluated against 604 manually created test cases, and it was shown that 56% of the test cases could be generated automatically by the SPECMATE pipeline.

Similar to the approaches by Thummalapenta et al. [51], Carvalho et al. [12] and Fischbach et al. [22] we also use the NLP levels of morphology, syntax, and semantics when processing user requirements. Unlike these approaches, we do not create executable test cases in this work; we create abstract test cases that require an instance of a data model to create executable test cases. In addition, we do not use any other formal representation except for representing the user requirements and the applications' data model as an enhanced user requirements model. This model is used as input to the transformation that produces the abstract test case model.

2.2.3 Model-Driven Engineering and Natural Language Processing

MDE and NLP: Gröpler et al. [25] present an approach for a machine-aided requirements formalization using NLP that supports the automatic generation of test cases. The pipeline used in this approach starts with user requirements written in natural language and transforms the requirements into a formal model (UML sequence diagrams) using semi-automated requirements formalization (ReForm). The pipeline then automatically transforms the sequence diagrams into UML state machines (ModGen) and automatically transforms the UML state machines into test cases (TCG). The novelty of this work lies in the first phase of the pipeline, which involves requirement formalization. This phase of the pipeline uses NLP tokenization, lemmatization, POS tagging, dependency parsing, and some semantic analysis. The syntactic entities are mapped to a predefined set of semantic entities. After NLP is applied to the requirements, the output is mapped to a domain-specific language - ifak requirements description language (IRDL). The textual representation of IRDL can then be transformed into a sequence diagram. The approach was evaluated using an industrial use case for charging approval of an electric vehicle interaction with a charging station. The evaluation metrics were based on the completeness, correctness, and consistency of the expected artifacts against the generated artifacts. The scores obtained without and with domain knowledge were completeness (79%, 82%), correctness (77%, 80%), and consistency (94%, 96%).

Wang et al. [52] describe their approach to automatically generating test cases from natural language specifications, referred to as Use Case Modeling for System-level Acceptance Tests Generation (UMTG). UMTG takes as input uses cases written in structured natural language (Restricted Use Case Modeling (RUCM)) and a domain model (a class diagram) and generates executable system test cases for acceptance testing. UMTG uses NLP to build Use Case Test Models (UCTMs) from

the RUCM specifications. UMTG uses five NLP analyses tokenization, named entity recognition, POS tagging, semantic role labeling and semantic similarity detection. NLP is used to extract the pre, post and guard conditions (constraints) needed to generate the test inputs for the test cases. OCL is used as an intermediate language to represent the constraints after applying NLP to the RUCM. Semantic role labeling is key to generating the OCL constraints that are then used to automatically generate test input data using constraint solving with a tool such as Alloy [30]. The empirical results using two industrial case studies show that UMTG can automatically and correctly generate 95% of the OCL constraints from the RUCM specifications, of which 99% of the constraints are correct.

Similar to the approaches by Gröpler et al. [25] and Wang et al. [52] we use NLP when processing the requirements employing similar NP analyses. We also require that use cases be written in structured natural language. Our approach is closer to that of Wang et al. [52] on the front end of the processing using NLP. That is, we use a data model that represents the data tables and relationships the application uses, whereas Wang et al. use a domain model represented as a class diagram. In addition, we implicitly capture the behavioral semantics of the use cases in the user requirements model used as input to the transformation to generate abstract test cases. Unlike both Gröpler et al. [25] and Wang et al. [52] we do not generate executable test cases in this phase of our work. The generation of test cases is left for future work using an instance of the data model and given specific coverage criteria.

CHAPTER 3

OVERVIEW OF APPROACH

In this chapter, we provide an overview of our approach to transforming user requirements to test cases. The research question being addressed here is: ***"How can the process of translating natural language requirements to test cases be further automated, given the recent advances in MDE and NLP?."*** The section 3.2 also introduces an example use case and the related entity-relationship model.

3.1 High-level Transformation

Figure 3.1 shows a high-level view of the transformation process from user requirements to test case generation. Figure 3.1 is updated from the previous published work [2] and uses additional models and processes in the transformation. The following section explains how each sub-problem is addressed in the figure 3.1,

1. SP-I: ***How can user requirements written as structured Natural Language along with the associated data model be represented in a semi-formal manner to support testing?*** is addressed by enumerated box's 1, 2, 3, 8, and 9(a).

2. SP-II: ***How can abstract test cases be generated given the semi-formal representation of user requirements model?*** is addressed by enumerated box's box 3, 4, and 5.

3. SP-III: ***How can concrete test cases be generated given abstract test cases and an instance of a data model derived from user requirements?*** is addressed by enumerated box's 5, 6, 7, and 9(b).

The following enumerated list describes in detail the overall transformation process.

(1) The source for the transformation process is either a use case (upper left side of Figure 3.1. 1.a - *Format 1: Use Case*) or a user story (bottom left side of the figure (b) - *Format 2: User Story*). The template used to submit the user requirement (use case or user story) is an instance of the meta-model defined in 2.a.

(2) *Requirements Meta-Model (a)* - represents the definition of the meta-model for the abstract syntax of the user requirement (b). A UML class diagram and Object Constraint Language (OCL) rules are used to define the meta-model. Details of the user requirements meta-model can be found in [2]. *Source Model (b)* - represents the use case or user story that conforms to the use requirements meta-model.

(3) *Enhanced Requirements Meta-Model (a)* - represents the definition of the meta-model for an extension of the user requirements that includes elements from the application's data model, referred to as the *Enhanced User Requirements Model (EURM)*. The *Enhanced User Requirements Model (b)* - is created by analyzing the user requirements using NLP and accessing an instance of the data model for the application.

(4) *Transformation Rules and Helpers (a)* - stores the transformation rules and helper functions needed to support the transformation process. The *Transformation Engine(b)* - uses the transformation rules and helper functions defined in (4.a) to transform the source model (3.b) to the target model (5.b) using a M2M transformation. We use M2M Atlas Transformation Language (ATL) [33].

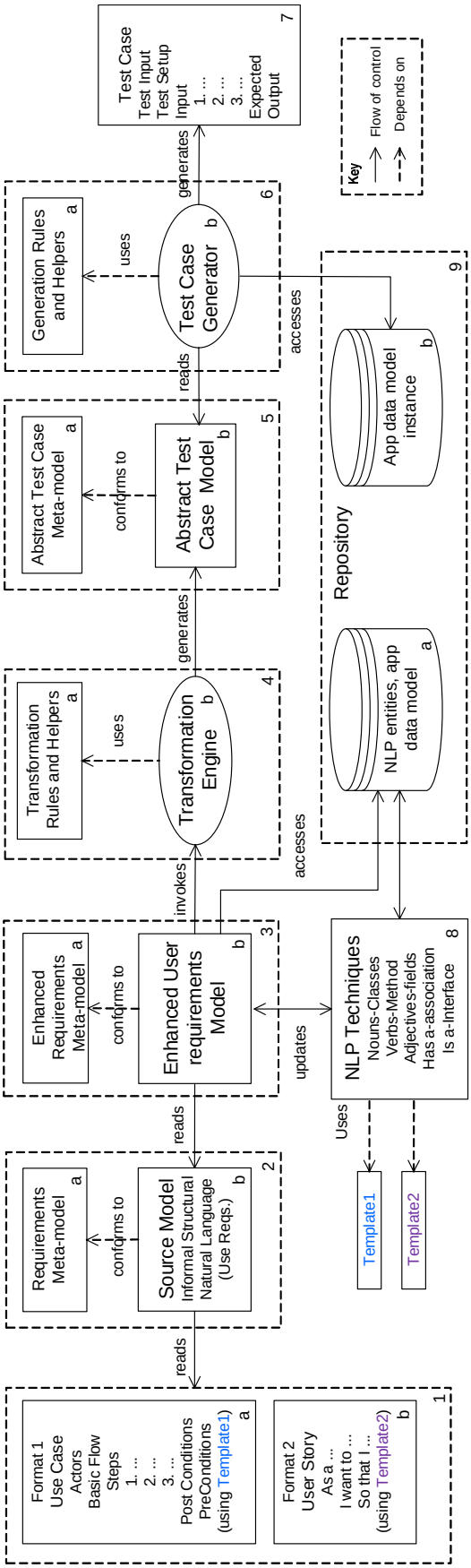

Figure 3.1: Overview of the high-level transformation process.

(5) *Abstract Testing Meta-Model (a)* - represents the definition of the meta-model for the abstract syntax for abstract test cases. *Abstract Test Case Model (b)* - represents an abstraction of the test case generated from the EURM. These abstract test cases are later instantiated to concrete test cases that can be executed by the system.

(6) *Generation Rules and Helpers (a)*- provides rules and helper functions needed to create concrete test cases. *Test Case Generator (b)* - generates multiple concrete test cases from an abstract test case by accessing an instance of the application's data model and using the rules and helper functions. Test case generation rules can be based on existing test generation techniques such as equivalence partitioning, or boundary value analysis [5].

(7) *Test Case* - represents the format of the final concrete test cases generated with all the data values.

(8) *NLP Techniques* - represents the algorithms and techniques required to process the user requirements. In this work, we used the Stanford Natural Language Processing toolkit [40] to process the user requirements.

(9) *Repository (a)* - contains the data model for the application and stores the intermediate model representations generated during NLP processing EURM model transformation. *Repository (b)* - contains instances of the application data model that are used during concrete test case generation.

3.2 Example User Requirement and Data Model

In this section, we introduce an illustrative example that includes a use case from an application named *Payroll Management System (PMS)* and the associated data model. PMS is used for calculating the pay of the employees at a company and was

Use Case ID: PMS-02-Login-Employer
Goal:
Employer logs into the system using their credentials
Actor: Employer
Precondition:
1. The Employer has an account
2. The Employer has the correct credentials
3. The Employer is on the login page
Description:
1. The Employer enters the username in the login page
2. The Employer enters the password in the login page
3. The Employer clicks on the sign_in button in the login page
Postcondition:
1. PMS system authenticates username and password
2. The Employer is redirected to the Employer homepage
3. The Employer can use PMS system functionalities

Figure 3.2: Example of the login use case for the Payroll Management System (PMS).

developed in a graduate software engineering class at Florida International University. PMS includes the following use cases login, logout, adding new employees, reporting hours worked, submitting employee time sheets, and approving employee time sheets.

As described in paper [2], a use case represents a piece of functionality in the system that accepts input from an actor and provides the actor with a result or may update the state of the system [31]. Use cases capture what the actor expects the system to do after the application is developed. We use a trivial use case as the example, that is, logging into the PMS system, as shown in Figure 3.2. We use a reduced format of the use case template that focuses mainly on the functionality of the transaction. In future work, we will include the entire format of the use case, including the non-functional requirements.

We decided to represent the data model for the system as an entity-relationship diagram (ERD). The Partial ERD for PMS is shown in Figure 3.3. The ERD in the figure shows a subset of the entities (5/9) used in the design of PMS. The 5 entities are `User`, `Employee`, `Employer`, `Department` and `Security Question`. The relationships

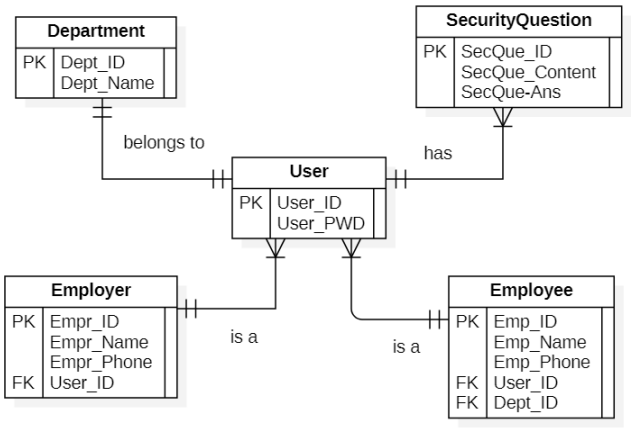

Figure 3.3: Partial entity-relation diagram (ERD) for PMS.

between the entities are described as follows. Each user belongs to a department, while each user is either an employee or employer, and every user has one or more security questions.

Figure 3.2 is used as a running example for future chapters. It is a continuously referred as usecase example throughout the dissertation that helps illustrate the concepts being discussed and provides a concrete demonstration of the ideas being presented. Both the 3.3 and 3.2 serves as a unifying thread throughout the dissertation and helps the reader to better understand the research work by providing a visual representation of the abstract concepts being discussed.

CHAPTER 4

CREATING AN ENHANCED USER REQUIREMENTS MODEL

In this chapter, we describe the meta-model for user requirements which consists of the meta-model for use cases and user stories, and how NLP processes the collected user requirements. Section 4.1 describes the meta-model for the user requirements representing both use cases and user stories. Section 4.2 presents the tool used to validate user requirements, and Section 4.3 presents a case study to show the output of the validation process. Section 4.4 details how the Enhanced User Requirements Model (EURM) is generated from the user requirements model, data model, and the entities generated during NLP.

4.1 Generic Meta-Model for User Requirements

As previously stated in Section 2.1.3, a meta-model can be thought of as a model that defines all instances of a domain type. In this section we describe the meta-model for user requirements, which can be thought of as a domain type consisting of either user cases or user stories. Tables 2.1 to 2.4 describe the properties of the user requirements domain type. Figure 4.1 shows the top level view of the meta-model

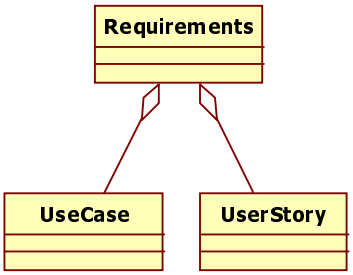

Figure 4.1: Top-level view of the meta-model for user requirements.

for user requirements (`UserRequirements`) which may be either a use case (`UseCase`) or a user story `UserStory`. The meta-model is illustrated using the ECore modeling language which is a subset of UML.

Figures 4.2 and 4.3 show in detail the meta-models for a use case and user story as described in the previous section. Most of the relationships between the classes shown in Figures 4.2 and 4.3 represent composition, which signifies ownership. For example in Figure 4.2, `UserCase` contains an `Actor`, a `FailedEndCondition`, a `SuccessEndCondition`, `NonFucntional` requirements and so on, which relate to the contents of the table shown in Tables 2.1 through 2.3. Similarly, Figure 4.3, `UserStory` contains an `Actor`, a `Function`, a `PostCondition` and an `Acceptance`, which relate to the contents of the table shown in Table 2.4.

Whenever a composition relationship is defined in the meta-model, a sequence of symbols, such as "[m..n]", appears at both ends of the relationship edge connecting the two classes. This implies multiplicity, which determines the acceptable number of instances the relation may have. Note that the attribute `descriptionback` in class `Acceptance` in Figure 4.3, has multiplicity of [0..1] which states that the attribute `descriptionback` may or may not have a value. This captures the notion that the acceptance criteria may be written on the back of the user story card, and may be used as an acceptance test if provided. The relationship edge connecting two classes is also annotated with names representing references in relationship. For example in Figure 4.3, the composition relationship between `UserStory` and `Actor` is annotated with the references `USactor` (user story actor) associated with the class `UserStory` and `actorUS` (actor user story) is associated with the class `Actor`.

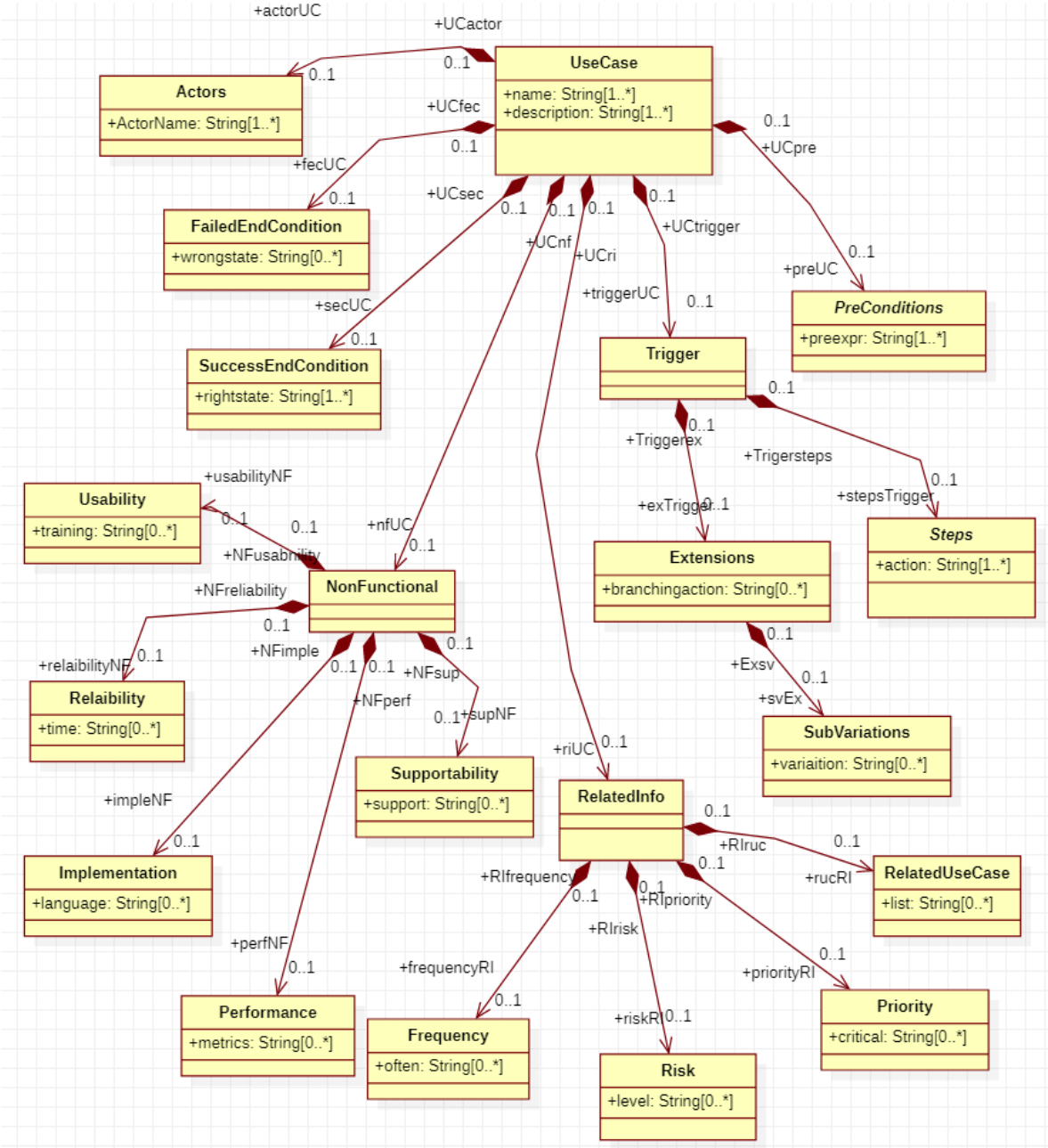

Figure 4.2: Meta-model for a use case.

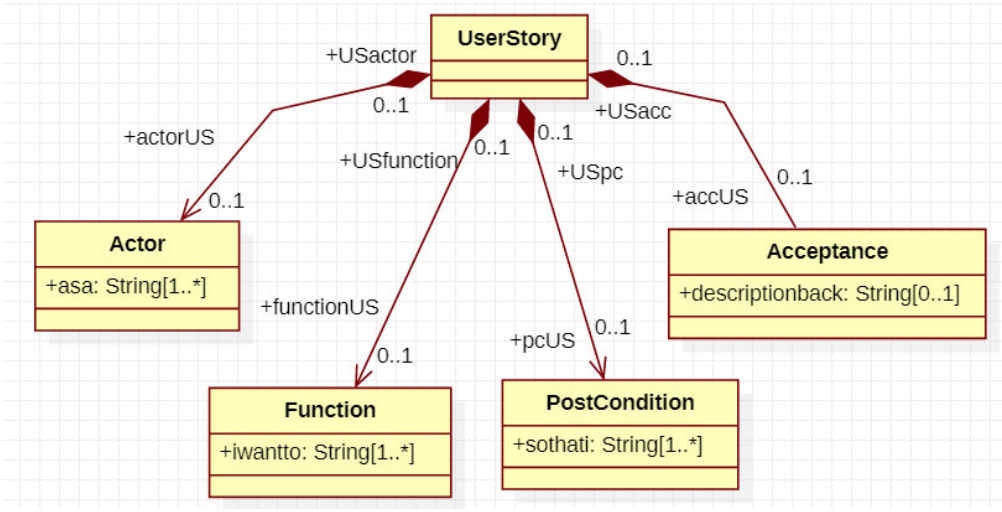

Figure 4.3: Meta-model for a user story.

4.2 Validation Tool for User Requirements

This section describes a tool that is used to process user requirements by first checking if the user requirements conform to the meta-model and then performing linguistic analysis on the user requirements. The main algorithm used in the tool is presented, followed by details of the implementation.

4.2.1 Architecture

Figure 4.4 shows the high-level architecture of the validation tool that checks conformity of user requirements against a meta-model and performs linguistic analysis of the user requirements' text. The processing done by the tool is shown in boxes 1, 2, 3, 8, 4 (partial), and 9 (partial) in Figure 3.1. The high-level architecture diagram in Figure 4.4 consists of 7 user-defined packages and 3 third-party applications. The third-party applications include the following. *EMF* (Eclipse Modeling Framework)

[19] is a modeling framework and code generation framework for building tools and other applications. EMF uses the Ecore modeling language. *OCLinEcore* [24] provides a textual concrete syntax that makes both Ecore and OCL accessible to users. *Stanford-NLP* [40] is a toolkit that provides an extensible pipeline that provides core natural language analysis.

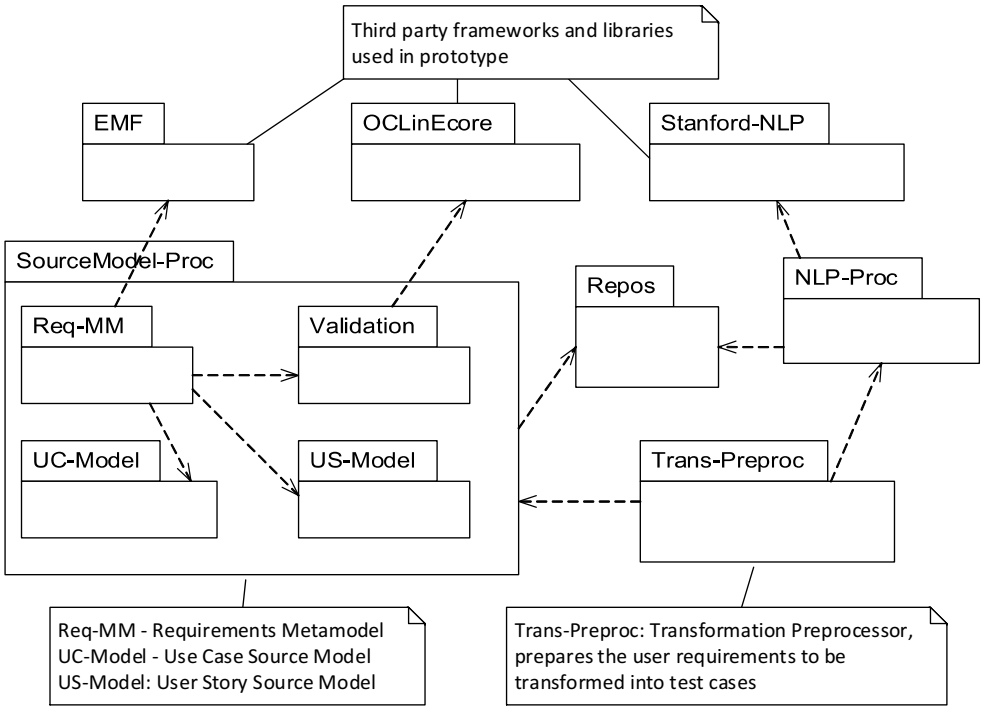

Figure 4.4: Architecture of tool to preprocess requirements models prior to transformation.

The user-defined packages are:

- `Req-MM` - Requirements meta-model defined in Ecore using EMF. This meta-model defines the meta-models for use cases and user stories.
- `UC-Model` - Reads use cases in the template provided. The use cases are structured based on the contents of Tables 2.1 to 2.3.
- `US-Model` - Reads user stories in the template provided. The user stories are structured based on the contents of Table 2.4.

- `Validation` - Validates the constraints written in the meta-model using OCL. If any constraints are violated, then error messages are generated and displayed.
- `NLP-Proc` - Analyzes the natural language text in user requirements by invoking user-defined rules on annotators that access the `Stanford-NLP` toolkit [40] to identify parts of speech and their relationships.
- `Repos` - Repository that stores validated models and NLP processed data.
- `Trans-Prerpoc`- Transformation engine pre-processor that coordinates the user requirements model validation and the linguistic analysis of the natural language text.

4.2.2 Implementation

The workflow of the validation tool using the architecture shown in Figure 4.4 starts in the `Trans-Prerpoc` package, reads in the user requirements models (use cases and user stories) into a container, then calls the `SourceModel-Proc` package to validate the user against the meta-model, shown in Figures 4.1, 4.2 and 4.3. The objects in the user requirements container use a structure similar to the meta-model.

The meta-model for user requirements is defined using the Eclipse Modeling Framework (EMF) [19], shown as a library at the top of Figure 4.4, and created using Ecore. The validation process includes using the `OCLinEcore` library, also shown at the top of Figure 4.4. During validation of the user requirement models, all errors and warnings are written to a log file and stored in the repository (`Repos`, shown in the center of Figure 4.4.

After the user requirements models are validated, they are then pre-processed using the pseudocode shown in Algorithm 4.1. The procedure named preproc_UR on Line 1 of the algorithm resides in the package `Trans-Prepoc` shown in Figure

Algorithm 4.1 Pre-processing of user requirements.

```
1: preproc_UR (ref URCon)
   /*Input:
   URCon - Container with user requirements either use cases or user stories
   Output:
   AnnURCon - Modified container with user requirements and processed text fields */
2: for each ur ∈ URCon do
3:    annur ← create()
4:    annur.actors ← ur.actors
5:    for each field ∈ {ur.fields}\{actors} do
6:       fieldSemans ← procNLP(field, annur.actors)
7:       annur.field.append(fieldSemans)
8:    end for
9:    AnnURCon.add(annur)
10: end for

11: procNLP (field, actors)
   /*Input:
   field - Text associated with the field of the user requirement
   actors - List of actors associated with the user requirement
   Output:
   semans - List of processed sentences identifying the parts of speech for each sentence in the field */
12: semans ← []
13: nlp_preproc ← NLP.augment(field, actors)
14: tagged ← NLP.pos(nlp_preproc)
15: for each rule ∈ KB.rules do
16:    seman ← NLP.apply(rule, tagged)
17:    semans.add(seman)
18: end for
19: return semans
```

4.4 and accesses the libraries provided by the `Stanford-NLP` tool-kit. The input to preproc_UR is the container with the user requirements models (`URCon`), and the output is a container with annotated user requirements models (`AnnURCon`). The for loop between Lines 2 and 8 iterates through each user requirement model (`ur`) in `URCon`, creates a new annotated user requirement object (`annur`), copies the actors from *ur* to `annur`, and for each field in *ur* call the NLP procedure `procNLP`).

The NLP procedure `procNLP` takes as input the field of the user requirement and the actors to the user requirement being processed. The output of `procNLP` is a list of semantics, each associated with a sentence in the text of the field. Each processed sentence identifies the nouns, verbs and adjectives in the context of the actors. The nouns represent the actors and other entities associated with the application, the verbs the action the actors take, and the adjectives the characteristics of the nouns.

4.2.3 Pre-Processing Example

To illustrate how the prototype works we present an abstraction of the results when the user story shown in Table 2.4 is pre-processed. The user requirement model, user story in this case, is first validated against the user story meta-model shown in Figures 4.1 and 4.3. The user story is validated against the meta-model and added to the container with the other user stories. After all the user stories have been validated this container is passed to the code whose algorithm is shown in Algorithm 4.1.

Figure 4.5 shows an abstract view of the results for the user story (Table 2.4) that is stored in the container with the annotated user requirements. Since our ultimate goal is to write a test case for the user story we are interested in identifying and POS tagged values that are related to the actors, the verbs which may signal and action by the actor, and other nouns and adjective that may be related to the actor. The top of the figure shows the relations that consists of <subject> <verb> <object>.

The dotted lines between the relations across the top of the figure shows an implicit hierarchy, that is the leftmost relation is the top of the hierarchy and the rightmost relation is at the bottom.

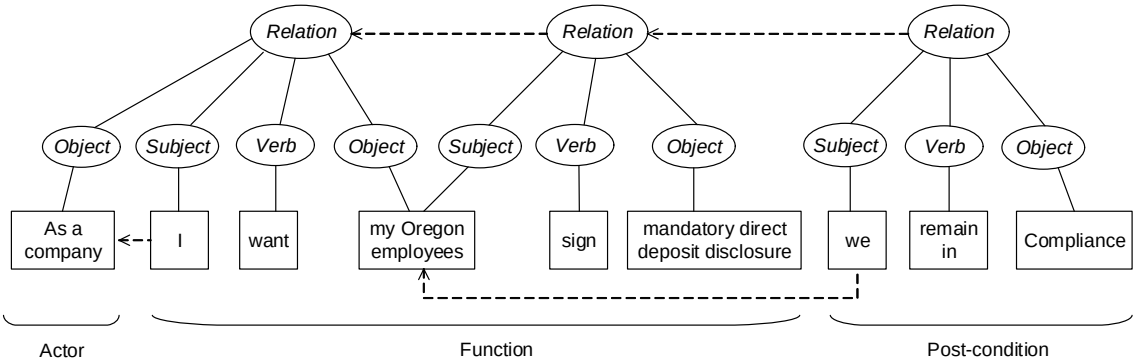

Figure 4.5: Partial tagging of user story model in Figure 2.4 using the open information extraction (open IE) technique [40].

Since we are using structured natural language for our user requirements we make use of our knowledge of the structure. In this example *company* is identified as the actor shown on the left of the figure, *I want* is associated with functionality of the actor, main target (noun) in the object of the functionality is *my Oregon employees*, which is also the subject for the main action (verb) of the functionality. The action is *sign*, the object of the action is *mandatory direct deposit disclosures*. In the post-condition the subject is *we* referring to *my Oregon employees*, the verb is *in* and the object if compliance. Note we did not show all the relations that re provided after the NLP processing of the user story.

Although not shown in Figure 4.5, we store the POS tags for each word in the user requirements. The other parts of speech we find interesting in the user story are: <nouns> = {*company, employees, deposit, disclosure, compliance*}; <proper nouns> = {*Oregon*}; and <adjectives> = {*mandatory, direct*} which are associated with *deposit disclosure*. Processing a use case is much more involved since there

are more fields in the use case and the text for each field is more involved. In the following section we present some metrics to show the complexity of using NLP in the pre-processing stage of the transformation.

4.3 Validation Tool Case Study

This section describes a case study that captures metrics by running the user requirements through the validation tool presented in Section 4.2. The user requirements include use cases from graduate student projects and user stories from an industry partner. The metrics presented include the number of errors and warnings for the user requirements and a summary of the POS tags and relations generated during NLP processing.

4.3.1 Setup

The user requirements for the experiments came from the projects of an undergraduate software engineering class at Florida International University and Anonymous Software Company. Table 4.1 shows the name of the application, the number of user requirements, and a brief description of the application. The use cases were from the CEN4010 Software Engineering 1 undergraduate class projects and the user stories from our Anonymous industry partner. An example of a project from the CEN4010 class is shown in row 1 of the table under the heading Use Cases shows the application *eMedi-Rec*, which is a medical record system for patients and doctors. The students in the class are required to use the following template: *Use Case ID*, *Goal*, *Precondition*, *Trigger*, *Description*, *Alternatives*, *Exceptions*, *Related Use Cases*, *Frequency*, *Criticality*, *Risk*, *Usability*, *Reliability*, *Performance*, *Supportability* and *Implementation*. The industry application is referred to as "HR1" and is from the human resources domain.

Table 4.1: User requirements (use cases and user stories) contained in the input for the experiments.

User Reqs.	# Reqs.	Description
Use Cases:		
eMedi-Rec	10	Medical record system for patients and doctors
IDB system	10	Inventory database for a Anonymous academic department
OPDB	10	Online Patient Database (OPDB) provides easy access to patient data at a clinic
Awesome	10	Project Awesome support movement of data between servers
F3T	10	Flex-Tic-Tac-Toe application web-based game for tic-tac-toe
User Stories:		
"HR1"	50	Application "HR1" is a human resources application

All user requirements are converted to XML Metadata Interchange (XMI) files using structures similar to those presented in Tables 2.1 through 2.4. XMI is a standard for exchanging metadata information developed by the Object Management Group [44]. The experiments are executed on a machine with a 2.5GHZ CPU, RAM with 8GB, and a hard disk with 1TB.

4.3.2 Results and Analysis

The results obtained from the experiments include metrics for the number of errors and warnings identified during the validation of the user requirements and a summary of the POS tags and relations generated during NLP processing. These metrics provide insight into the additional processing required to transform the user requirements into test cases using the approach presented in Figure 3.1. Table 4.2 shows the results of the validation process of the user requirements using the meta-model shown in Figures 4.1, 4.2, and 4.3. Row 1 in the Use Cases section of Table 4.2 showed

that eMedic-Rec application produced 0 errors and 10 warnings. The errors represent missing functional requirements essential for the M2M transformation, and the warnings are for missing related information and non-functional requirements. In the case of eMedic-Rec the warnings were generated due to the missing failure condition in each use case.

Table 4.2: Validation of user requirements against meta-model.

User Reqs.	Errors	Warnings	Explanation of Errors
Use Cases:			
eMedi-Rec	0	10	No failure postcondition presented.
IDB system	4	0	Missed use case descriptions
OPDB	0	0	-
Awesome	0	10	No related use cases mentioned
F3T	0	10	Missed implementation in non-fuctional requirements
User Stories:			
"HR1"	4	20	Errors: No postconditions Warnings: No acceptance test.

Table 4.3 shows the data generated during the NLP processing of the user requirements. Row 1 of the table shows that 10 use cases for eMedic-Rec have 210 Subject Relation Object (SRO) relations tuples, 776 Nouns, 431 Verbs, 84 Adverbs, and 264 Adjectives. The 50 user stories in the "HR1" application had a total of 340 SRO relation tuples, 723 Nouns, 444 Verbs, 88 Adverbs, and 304 Adjectives. On average, the use cases contained more POS tags than the user stories as shown in the last row of the use case and user stories, respectively. Table 4.4 shows an estimate of the average time needed to process the user requirements (50 use cases and 50 user stories). The NLP processing takes approximately 45 times longer than the validation process.

Table 4.3: NLP analysis of user requirements.

User Reqs.	Relations (SRO)	Nouns	Verbs	Adverbs	Adjectives
Use Cases:					
eMedi-Rec	210	776	431	84	264
IDB system	719	1374	700	133	460
OPDB	394	732	356	76	178
Awesome	241	742	362	85	197
F3T	209	794	471	97	264
Averages:	35.5	88.4	46.4	9.5	27.3
User Stories:					
"HR1"	340	723	444	88	304
Averages:	6.8	14.5	8.9	1.8	6.1

Table 4.4: Average time recorded to validate user requirements and NLP processing.

User Reqs.	Total	Meta-Model Processing (msec.)	NLP Processing (sec.)
Use Cases	50	<1.0	51.6
User Stories	50	<1.0	45.1

4.4 Enhanced User Requirements Model (EURM)

This section describes how the enhanced user requirements model (EURM) is created. To create an EURM, we define an EUR meta-model and an algorithm to create the EURM from the user requirement model.

4.4.1 Enhanced User Requirements Meta-Model

To support the transformation process shown in Figure 3.1, boxes 3, 4, and 5, we define a meta-model for the EURM. This meta-model is shown in the box labeled 3(a) in Figure 3.1. A partial representation of the enhanced user requirements meta-model, referred to as `EUR-MM`, is shown in Figure 4.6. `EUR-MM` consists of `UserReq` -

the user requirements meta-model derived from the meta-model described in Allala et al. [2] and a `DataModel` that represents parts of the meta-model for an enhanced entity-relation model [20].

The `UserReq`, shown in Figure 4.6 describes the meta-model for the use cases, and consists of a `Goal` - a summary of the intent of the use case; `Actor` - external entity interacting with the system; `Precondition` - constraints that should hold before the functionality in the use cases is performed; `Description` - the flow (`Flow`) of steps (`Step`) representing the functionality; and `Postcondition` - the constraints that should hold after execution of the flow representing the functionality. The `DataModel` consists of one or more entities (tables) in the data model. Each `DM_Entity` consists of one or more table attributes (`DM_Attribute`). The concept of an alias (`Alias`) is introduced into the meta-model that connects various terms used in the use case to the actual names of entities and attributes in the data model. The `SysEntity` class refers to different elements of the actual system, such as pages, page fields, and sessions.

Other classes shown in Figure 4.6 are the classes representing the components of a constraint and the components of a step in the flow of the functionality for the use case. A constraint is composed of the entities (`CEntity`) and an operator (`COperator`) that may be a combination of logical, relational, and membership applied to entities. Each step in the flow process is composed of `SSubject` - the main actor in the step, `SAction` the action performed by the subject, `SObject` - the receiver of the action, and optionally `SDest` - the intended target of the action. The classes described in this paragraph may all have aliases related to elements of the data model or other elements of the system, e.g., an attribute in a data model entity, a page, or a field in a page. An instance of the EUR meta-model in Figure 4.6 is shown on the left side of Figure 5.3.

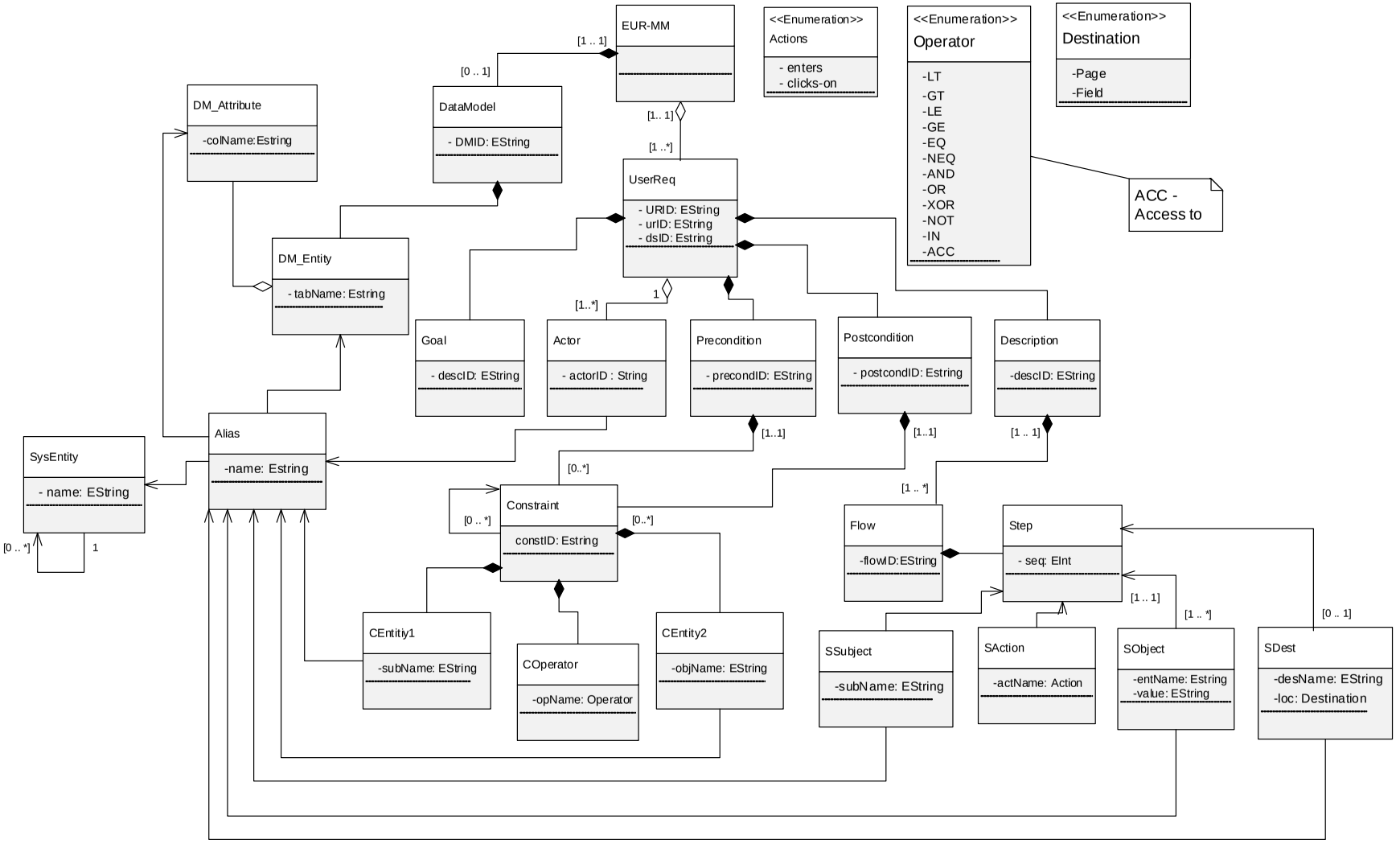

Figure 4.6: Partial enhanced user requirements (EUR) meta-model.

4.4.2 Generating the EURM

In this section, we describe how user requirements are processed using NLP-based techniques, see box 8, and the data model in the repository, see the cylinder labeled "a" in box 9 in Figure 3.1. After we define the structure of a model that conforms to a meta-model, we apply NLP techniques to perform linguistic analysis on the user requirements text. The NLP processing workflow uses features from the Stanford CoreNLP suite [40] such as the tokenizer, sentence splitter, POS tagger, and dependency parser. After NLP processing, there is still some ambiguity regarding the context and POS tagging of words. We use manual input in cases where a value cannot be identified when applying our matching procedure. A knowledge base is created to keep track of the user input and maps phrases used in the EURM to the phrases used in the user requirements. To support the algorithm we have created a sub string matching approach see section 4.4.3for detailed understanding.

Algorithm 7.1 in Appendix B shows the details of the algorithm used to process a user requirements model (URM) and create an enhanced user requirements model (EURM). The algorithm consists of two procedures (1) the main, `gen_EURM`, see line 1, and (2) the preprocessor, `preproc_URM`, see line 39. Both procedures take in two parameters, `URM` and `Repos` - a reference to the repository containing all the processed NLP entities, the application data model, and associated graphs. Algorithm 7.1 in Appendix B extends the NLP approach done in Algorithm 4.1.

The pre-processing procedure `preproc_URM`, lines 39 to 51, performs the NLP activities described in the first paragraph of this section. For each entry in the URM and annotated POS, `anpos`, is created, which is then sent through the pipeline to create annotated objects, `anobjs`, for the entry. These objects are then processed to generate annotated subject, relation, and object (SRO) triples, `ansros`, and added to the repository along with the entry. The pre-processing procedure also creates a

bidirectional graph for the data model, line 49, which is later used to create aliases for the elements of the URM.

The main procedure starting at line 1 produces a EURM and the updated repository as output.. This procedure starts by creating a EURM, `eurm`, an instance of the EUR meta-model containing only its structure, see line 2. As each component of the EURM is processed, it is updated with the processed UR entry details, e.g., lines 6, 9, 15, 23, and 32. In line 3, the pre-processing procedure is called. Lines 5 through 10 checks if the entry is a use case id, `UC_ID` or a goal, `GOAL` and add them to the EURM unchanged.

Lines 11 to 16 process the entry of type actor, `ACTOR`. This involves calling the procedure `matchingActor` that takes the UR entry and repository as parameters. All the matching procedures do a fuzzy substring matching to see if entries in the UR entry match entities in the data model (`dmgraph`) and phrases stored in the knowledge base for the EURM. If the threshold is satisfied, the match is done automatically. Otherwise, user input is required, the EURM is updated, and the mapping is added to the knowledge base. Currently, we have a high threshold since determining the appropriate threshold will be considered in future work. The matching algorithms for the precondition, description, and postcondition are all tailored based on the structure of the EURM meta-model. For example, some phrases used in the URM may need to be replaced by phrases used in the knowledge base of the EURM.

Lines 17 to 25 show the processing performed on each entry in the precondition component of the UR model. The `matchingDescript` procedure returns a list consisting of `CEntry1`, `COperator` and `CEntry2` as a JSON-like string stored in `matchDescript`. For each precondition entry, the POS and SRO nouns are checked, and fuzzy substring matching is performed on the data model to create the `CEntity1`. For the `COperator`, the POS verbs (e.g., has, have, clicks, etc.) are checked, followed

by the SRO relations. A POS verb and SRO relation match mean the same word occurs in the knowledge base for constraint operators, ignoring case or a subset, e.g., enter or enters. The fuzzy substring function is invoked if there is no direct match. In the case of duplicates, the most frequent match is selected. If no match is made, the system asks for a manual review. The processing for the `CEntry2` is similar to the `CEntry1` except the focus is on nouns and adjectives.

The processing of the description component (`DESCRIPT`) of the UR model, lines 26 to 34, is similar to the precondition, except there are four parts to be generated, `SSubject`, `SAction`, `SObject`, and `SDest`, see Section 4.4.1 for a description of these parts. The processing of the postcondition component (`POSTCOND`), see lines 35 to 37, is similar to that of the precondition, lines 17 to 25. An EURM for the user requirement is in Figure 4.7 and is also shown on the left side of Figure 5.3.

4.4.3 Fuzzy String Search

Fuzzy string search is a technique used to find strings that are approximately similar to a given pattern or query string, even when there are differences in spelling, typos, or other inaccuracies [34]. It is particularly useful when searching for information in large databases where the search query may not match exactly with the data in the database.

Fuzzy string search algorithms typically use string metrics, such as the Levenshtein distance, to calculate the difference between the query string and each candidate string in the database. The Levenshtein distance is a measure of the difference between two strings, and it considers the number of edit operations (insertions, deletions, and substitutions) required to transform one string into the other. The algorithms can be configured to return only the top results or all results within a certain distance from the query string. Some fuzzy string search algorithms also allow for the use of

weighting factors, which can be used to give more importance to certain parts of the strings being compared. An example of a fuzzy string search can be finding similar words to a misspelled word in a dictionary. For instance, if the user searches for "pwd" instead of "password," a fuzzy string search algorithm would return "password" as a close match.

Fuzzy string search algorithms are used in a variety of applications, including spell-checking, text correction, information retrieval, and database search. They are beneficial when working with large amounts of text data, where typos and other inaccuracies are common, which helps address our problem of inconsistency. It's a technique used to find strings that are approximately similar to a given pattern or query string, even when there are differences in spelling, typos, or other inaccuracies. Fuzzy string search algorithms use string metrics, such as the Levenshtein distance, to calculate the difference between the query string and each candidate string in the database and to rank candidate strings based on their similarity to the query string.

4.4.4 Illustrative Example of Generating an EURM

Figure 4.7 shows an example of an EURM derived from a user requirements model (URM), see Figure 3.2, and the data model, see Figure 3.3. On the left side of the figure, we show the URM that adheres to the requirements meta-model shown in Figure 4.1. The right side shows the EURM generated using the derivation process described in this section, Boxes 2, 3, 8, and 9 in Figure 3.1.

Starting at the top of Figure 4.7, we leave `Use_Case_ID` as is and move on to processing the `Actor`. In our use case example, the Employer is the `Actor` and has a similar name in our data model. We use the algorithm in Section 4.4.2 to connect Employer to another entity, `User`, which further consists of one or more attributes (`DM_Attribute`), including `User_ID` and `User_PWD`.

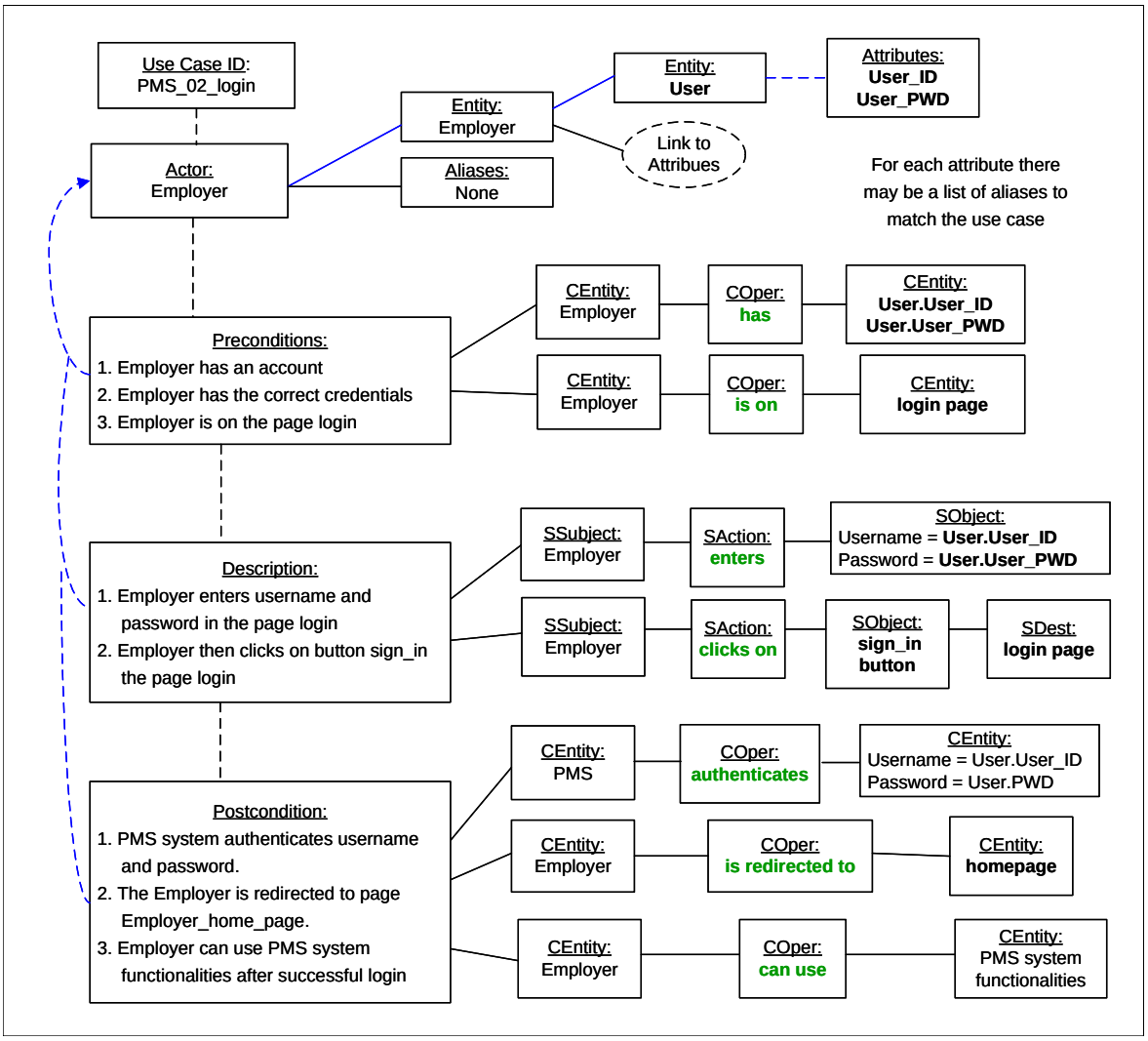

Figure 4.7: Illustration showing the enhanced user requirements model derived from a use case login model.

We continue by analyzing the `Precondition` using the algorithm, NLP, and the data model stored in the repository to generate the component for each sentence/phrase in the precondition. In this example, each entry in the `Precondition` is analyzed to produce a `CEntity`, `COper`, and `CEntity`. The first entry in the precondition (*Employer has account*) is translated to mean that there are valid credentials for the employer in the data model. Note that if the algorithm cannot make such a determination, it is manually analyzed, and a interpretation is provided.

A similar approach is used to analyze each entry in the `Description` component of the URM. That is, the `SSubject`, `SAction`, `SObject`, and `SDest` (optional) are generated from (*1. Employer enters username and password into the page login*). This analysis results in `SSubject = Employer`, `SAction = enters`, and `SObject = {Username = User.User_ID, Password = User.User_PWD}`. The second entry in the precondition is similarly processed except there is an `SDest = page login`. A similar approach to transforming the precondition is used for the postcondition. The main difference is that certain keywords are translated into a system state.

4.5 Summary of Chapter

This chapter presents the solution for SP-I of our research question. The approach for the solution includes creating an enhanced user requirements model (EURM) from commonly used user requirement models (URMs). The first stage is creating a meta-model for URMs and validating that the meta-model is correct. The next stage is to extend the meta-model for URMs to include elements of the data model for a specific application and results of NLP performed on the text of the user requirements. Algorithms are presented that support the process of creating EURMs from URMs. To validate the correctness of the user requirements meta-model, we developed a tool that takes the URM as input and generates feedback to the user in terms of warnings and errors. A case study is presented that validates the user requirements for 50 use cases from undergraduate student projects and 50 use stories from an industrial application. Overall the results were positive, showing that the meta-model is close to being correct. The meta-model for EURM is validated using experiments performed in the following chapter when transforming EURMs to abstract test cases.

CHAPTER 5

TRANSFORMING ENHANCED USER REQUIREMENTS MODELS TO ABSTRACT TEST CASES

This chapter describes how enhanced user requirements models (EURMs) are transformed into abstract test cases (ATCs) and the approach to validate the transformation process. Section 5.1 presents details of the ATC meta-model and the ATLAS Transformation Language (ATL) [33] rules used during the transformation process. Section 5.2 describes the transformation engine built to transform EURMs into ATCs. The chapter concludes in Section 5.3 with a case study showing how the transformation engine was validated using use cases from a graduate software engineering project and undergraduate software testing projects at Florida International University. The transformation process is shown in the boxes labeled 3, 4, and 5 in Figure 3.1. This chapter presents a solution to ***SP-II:***: How can abstract test cases be generated given the semi-formal representation of user requirements model?

5.1 Transforming EURs to Abstract Test Cases

In this section we describe how EURMs are transformed into ATCs. The activities needed to realize the transformation process include the creation of an ATC meta-model and a set of transformation rules, written in ATL, between the EURM meta-model and the ATC meta-model.

5.1.1 Abstract Test Case Meta-model

Figure 5.1 shows the partial ATC meta-model. The meta-model, `ATC-MM`, consists of an abstract test case `ATC` and a data model `DataModel`. The data model is similar to the one used in the EUR meta-model, shown in Figure 4.6. The ATC meta-model

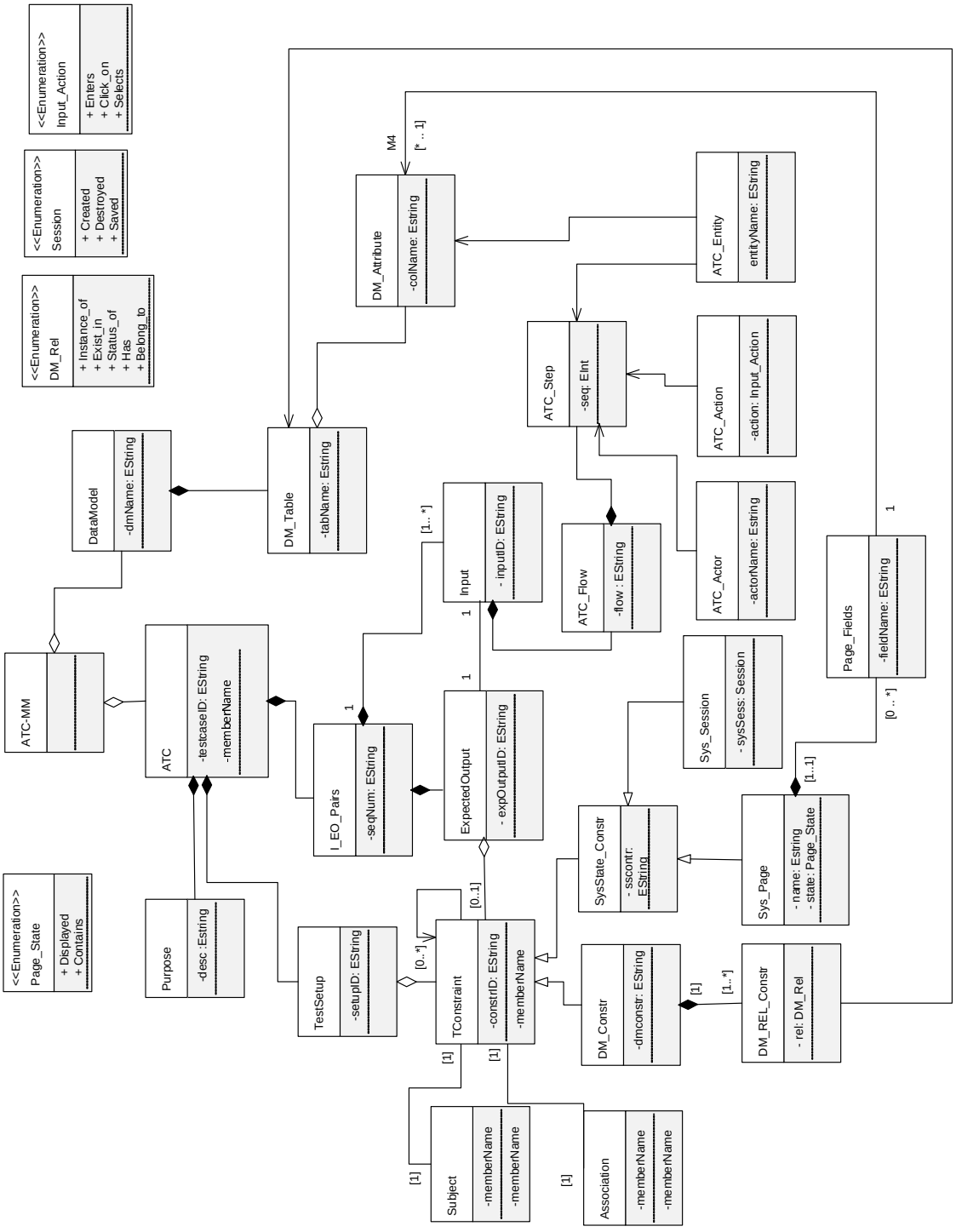

Figure 5.1: Partial abstract test case meta-model.

consists of `Purpose` - a description of the ATC, `TestSetup` - that state the system should be in for the successful execution of the test case, and `I_EO_Pairs` - the inputs and the expected output of the test case. The test setup and expected outputs are both defined as constraints (`TConstraint`). These constraints may be either a data model constraint (`DM_Constr`) or a system state constraint (`SysState_Constr`). Each ATC input consists of a flow (`ATC_Flow`) and steps (`ATC_Step`) similar to the EUR meta-model. The ATC meta-model currently includes four enumeration types `DM_Operators` - the operators related to the data model, `Page_State` - the state of the page the actor is accessing, `Session_State` - the current state of the system, and `Input_Action` - the action taken by the actor.

Expanding on the explanation of Figure 5.1 are the classes representing the components of the data model constraint, system state constraint, and the step for the flow of the test case input. The data model constraint `DM_Constr` is composed of a data model entity, data model operator (`DM_Operator`), and data model attribute. The system state constraint consists of a page state or a session. Similar to the data model operator, the page state and session state are enumerated types shown at the top of Figure 5.1. Note, for each enumerated type we show the initial set, which may be expanded upon in future work. There is an association between the fields on the page and the attributes in the data model. The step in the input flow consists of an actor (`ATC_Actor`), action (`ATC_Action`), and the entity receiving the action (`ATC_Entity`), which may be a page in the system. An instance of the ATC meta-model in Figure 5.1 is shown on the right side of Figure 5.3.

5.1.2 Transformation Rules

An ATL [33] model-to-model transformation is composed of a set of transformation rules and helpers that handles the mapping between the source meta-model (EUR)

elements, shown in Figure 4.6 and the target meta-model (ATC) elements, shown in Figure 5.1. Each rule specifies how the source model elements must be matched and navigated in order to initialize target model elements. Typically, ATL model transformations are composed of declarative rules, referred to as matched rules. ATL is based on OCL [43] for its data types and declarative expressions. The transformation engine automatically executes these rules for every match in the source model according to the source patterns of the matched rules. The transformation process uses declarative rules for `testcaseID`, `purpose`, and so on. There also exist rules that have to be explicitly called from another rule, namely lazy rules, which give more control over the transformation process. Each rule contains a unique name.

Figure 5.2 shows several rules used in the transformation engine. In this example, the execution starts on line 23 with the rule `Precondition2Testsetup`, which iterates through each precondition entry and generates an entry for test setup by calling the lazy rule `Constraints2TConstraint`, line 6. The `Constraints2TConstraint` rule then transforms each component of the precondition entry into its counterpart in the test setup entry, starting with the component subject. The precondition subject is transformed using the lazy rule `CSubject2Subject`. The source pattern used in each rule is shown after the keyword `from`, and the target pattern is shown after the keyword `to`. The source and target patterns are uniquely labeled, usually with a lowercase letter, e.g., `a`, before the colon (`:`). Note that there can be multiple target patterns for a rule. The left arrow (←) signifies the binding between the result of an expression and a feature name. The expression on the right side of the left arrow may be an OCL expression that processes a collection, see line 27 in Figure 5.2. ATL can manipulate collections similar to OCL. The right side of the left arrow may also use helper functions to assist with processing collections.

```
1. helper context EUR!Precondition def: allclasses():
2. Sequence(EUR!Precondition)=
3.  self.conprecon->iterate(e;
4.    acc:Sequence(EUR!Precondition)=Sequence{}|acc->union(Set{e}));
5.
6. lazy rule Constraints2TConstraint{
7. from
8.       e:EUR!Constraints(true)
9. to
10.      a:ATC!DM_Constr(subject<-thisModule.CSubject2Subject(e),
11.          em_rel_constr<-thisModule.CRelation2DM_REL_Constr(e),
12.          dm_table<-thisModule.TS2DM_Table(e))
13. }
14.
15. lazy rule CSubject2Subject{
16. from
17.      e:EUR!Constraints(true)
18.  to
19.      x:ATC!Subject(subject<-e.csubcon.cSubject),
20.      y:ATC!DM_Constr()
21.  }
22.
23. rule Precondition2Testsetup{
24. from
25.      e:EUR!Precondition(true)
26. to
27.      a:ATC!TestSetup(tconstraint<- e.allclasses()->collect(a|
28.       thisModule.Constraints2TConstraint(a)))
29.  }
```

Figure 5.2: Atlas transformation language.

Helper functions can be used to define (global) variables and functions. They can call each other (recursion is possible) or be called from within the rules. In the example in Figure 5.2 we define a helper function, lines 1 to 4, which iterates over each precondition entry, and this allows the target pattern `ATC!TestSetup` to invoke the rule `Constraints2TConstraint` for each precondition entry. The entire set of ATL rules is shown in Appendix , Figures A10 to A13.

5.1.3 Illustrative Example of Transformation

Figure 5.3 shows an example of the transformation from EURM to ATC. On the left side of the figure we show the EURM that is created from the use case shown

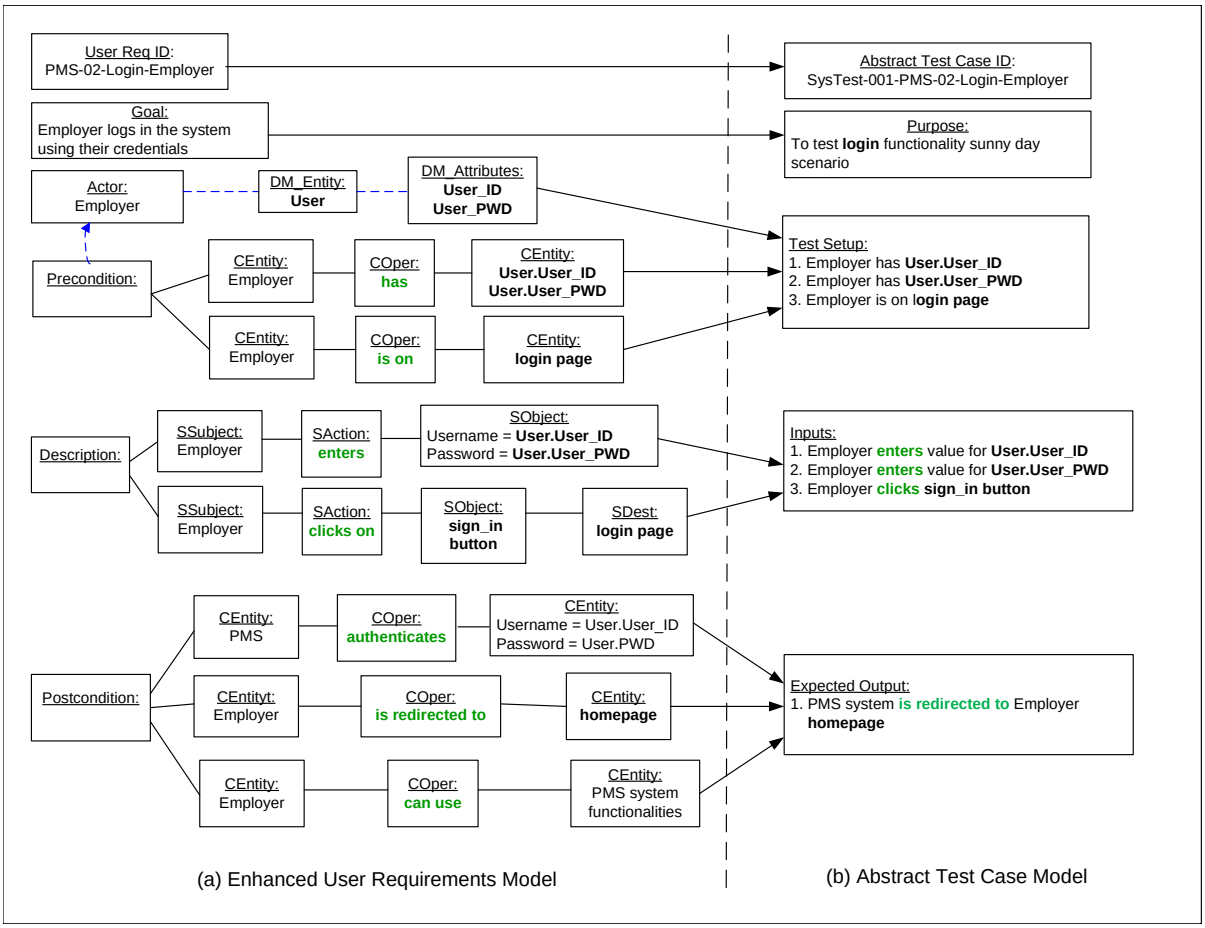

Figure 5.3: Illustration showing the transformation of an enhanced user requirements model to an abstract test case model.

Figure 3.2. The right side shows the ATC generated using the transformation process described in this section, Boxes 3, 4, and 5 in Figure 3.1.

Starting at the top of Figure 5.3, we transform `User Req ID` to the `Abstract Test Case ID` and the `Goal` to the `Purpose` with small changes to the text. The `Actor` in our caseis Employer and is represented as `User` in our data model we use the above algorithm in Section 4.4.2 to connect the entity `DM Entity` which further consists of one or more attributes (`DM-Attribute`) that are `User ID` and `User PWD`.

We use a combination of `Actor` and `Precondition` to generate the entries for `Test Setup`. In this example each entry in the `Test Setup` is generated from the

components of the precondition `CEntity`, `COper`, and `CEntity`. The first entry in the precondition (*Employer has User.User_ID and User.User_PWD*) is transformed into the first two entries of the test setup (*1. Employer has User.User_ID* and *2. Employer has User.User_PWD*).

A similar approach is used to transform each entry in the `Description` component of the EURM. That is, the `SSubject`, `SAction` and `SObject` representing *1. Employer enters User.User_ID and Password = User.User_PWD* is transformed into the `Inputs` entries (*1. User enters value for User.User_ID*) and (*2. Employer enters value for User.User_PWD*). A similar approach to transforming the precondition is used for the postcondition. The main difference is that certain keywords are translated into system state.

5.2 Transformation Engine

In this section, we describe the transformation engine that converts valid EURMs into ATCs. The EURM is generated from the user requirements and validated against the EURM meta-model. Using the transformation rules written in ATL [33], a EURM is transformed into an ATC.

5.2.1 Architecture

Figure 5.4 shows the high-level architecture of the engine that defines the transformation process. The processing done by the engine is shown in boxes 3, 4, and 5 in Figure 3.1. The high-level architecture diagram in Figure 5.4 consists of 5 user-defined packages and 3 third-party applications. The third party applications include *EMF* (Eclipse Modeling Framework) [19], *OCLinEcore* [24] and *Stanford-NLP* [40] previously described in Section 4.2. Since the engine includes some packages from the

validation tool, Figure 4.4, we briefly describe the new packages in the transformation engine.

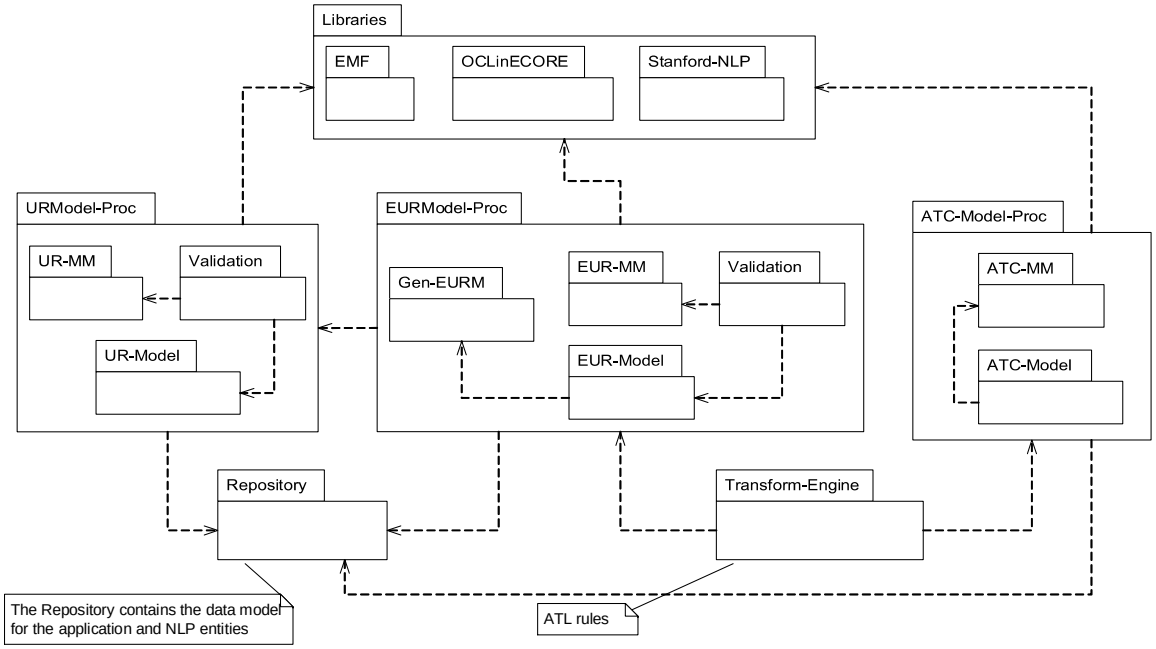

Figure 5.4: High-level architecture of the transformation engine.

The new user-defined packages are:

- `EUR-MM`- Enhanced User Requirements meta-model defined in Ecore using EMF, see Figure 4.6 and Appendix , Figures A2 to A4.

- `EUR-Model`- Read the generated EUR model generated by the algorithm using the user requirements model and NLP entities stored in the repository.

- `Transformation Engine` - Contains the rules and helper functions written in ATL required to perform the transformation between EURMs and ATCs. The engine accesses various artifacts from the `EURModel-Proc` and generates an ATC model for the `ATC-Model-Proc`.

- `ATC-MM`- Abstract Test Case meta-model defined in Ecore using EMF. This meta-model ensures that ATC models are developed using the correct syntax and semantics.

- `ATC-Model`- Stores the ATC model generated by the engine while it is being created. This model is then stored in the repository and later used to create concrete test cases. The generation of concrete test cases is described in Chapter 6.

5.2.2 Implementation

The workflow of the transformation engine using the architecture shown in Figure 5.4 starts in the `EURModel-Proc` package, reads in the enhanced user requirements model into a container, then calls the `EUR-MM` to validate the EURM against the meta-model, shown in Figures 4.6. The objects in the EUR-Model container use a structure similar to the meta-model.

The EURMs are generated from user requirements models first by invoking the Algorithm 7.1 to perform processing of the user requirements, then invoking Algorithm 4.1 to perform pre-processing of the EURM before it is passed to the transformation engine. The validation processes for both the user requirements model and the EURM are done by using the `OCLinEcore` library shown at the top of Figure 5.4. During validation of the EURM all errors and warnings will prohibit further automation and require a prompt for human interaction.

After the EURMs are validated, they are then used as input to the transformation process. The procedures defined in the `Transform-Engine` take the input from the `EUR-MM` and `EUR-Model` and generates an `ATC-Model`, which is consistent with the ATC meta-model in `ATC-MM`.

5.3 Case Study

In this section, we present a case study showing the feasibility and practicality of transforming EURMS into ATCs. The case study is also used to validate the approach by automatically generating abstract test cases from use cases and comparing the results to the manually generated abstract test cases. The research questions being investigated are: (1) *How similar are the abstract test cases (ATCs) generated using the MDSE and NLP automated approach compared to the ATCs generated using the manual approach?* (2) *What are the major limitations of using the MDSE and NLP automated approach to generate ATCs?* We discuss the results of the case study in the context of these questions.

5.3.1 Setup

As described in Section 3.2, the user requirements for the experiments came from a project from a graduate software engineering class at Florida International University (FIU) in the Fall of 2015. The application's name is the *Payroll Management System (PMS)* and is used to calculate the employees' pay at a company. The students were required to submit three deliverables, including a requirements specification document, a design document, and a final document that included unit, subsystem, and system test cases. The student project consisted of 28 use cases, of which 12 use cases were implemented for the PMS application. Table 5.1 shows the user requirements for the 12 use cases used in the case study. The columns from left to right in the table are row number, use case id, and the goal of the use case. One of these use cases, shown in Row 1 of the table (`PMS-01-Login-Employee`), was the illustrative example presented throughout the dissertation.

The abstract test cases (ATCs) used in the case study were automatically generated using the approach shown in Figure 3.1 and manually by the authors of Allala

et al. [3], independent of the lead author who implemented the automated system. Additional details on the process of how the ATCs were manually generated are presented in Section 5.3.2. To manually generate the ATCs, we reviewed the system test cases from the PMS graduate project and four undergraduate software testing projects at FIU, two projects from Summer 2018 and two projects from Spring 2020. The goal of the undergraduate testing projects was to perform unit, subsystem, and system testing on PMS. Students in the undergraduate class were required to submit two deliverables, the first deliverable using only black-box testing techniques [4] and the second using white-box testing techniques [4] to improve the test coverage. We used the test cases from the second deliverable in this project.

Table 5.1: User requirements (use cases) contained in the input for the experiments.

Use Cases		
#	ID	Goal
1	PMS-01-Login-Employee	Employee enters his/her username and password to login to the system
2	PMS-02-Login-Employer	Employer enters his/her username and password to login to the system
3	PMS-210-Logout-Employee	Employee clicks on logout button to logout of the system
4	PMS-211-Logout-Employer	Employer clicks on logout button to logout of the system
5	PMS-05-ApproveTS	The employer approves the Employee's timesheet submitted information
6	PMS-04-ModifyTS	The employer modifies the Employee's timesheet information
7	PMS-13-AddEmployee	The Employer adds a new Employee to the system by visiting the Manage tab
8	PMS-03-SearchEmp	The Employer searches for the Employee's information
9	PMS-08-SaveTS	Employer clicks on logout button to logout of the system
10	PMS-06-CalPay	Employer request that the system calculate the pay for the Employee
11	PMS-09-PayCheck	The Employee views their paycheck
12	PMS-07-ChangePwd-Employee	The Employee changes their password

5.3.2 Processing ATCs

Manually Generating ATCs: The research team, excluding the person implementing the automated approach, generated one ATC using system test cases from five projects. These projects included the PMS graduate student project and the four undergraduate student projects. Therefore to generate one ATC, at least five test cases were reviewed. Before creating the ATCs, the team members reviewed the complete meta-model for ATCs. A partial meta-model is shown in Figure 5.1. After the team members understood the meta-model and what instances of an ATC model looked like, two team members working independently generated their ATCs. Generating an ATC involved each member reviewing their five test cases and the data model instances (databases) from the various projects and reverse engineering them to get the ATC. After the generation of the ATCs was completed for each of the 12 use cases in Table 5.2, the two team members compared their ATCs for any discrepancies. Discrepancies were resolved by a third team member, independent of the lead author of [3], to create a single ATC.

Automatically Generating ATCs: The transformation engine described in Section 5.2 was used to generate the ATCs from user requirements, similar to the use case shown in Figure 3.2. The engine was built using a pipe-and-filter architecture where the source was the XML representation of the use case and the sink the XML representation of the ATC. The intermediate models generated in the prototype used the XML Metadata Interchange (XMI), a standard for exchanging metadata information [44]. These intermediate models were manually checked for correctness against the various meta-models used in the transformation process.

Comparing ATCs: The comparison of the manually and automatically generated ATCs was done based on the structure of the meta-model for ATCs in Figure 5.1. The entries for *Test Case ID*, *Purpose*, *Test Setup*, *Inputs*, and *Expected Outputs* were

compared for similarities. Comparing the entries for *Test Setup*, *Inputs*, and *Expected Outputs* was more complicated than the other parts of the ATC. These components include references to the PMS data model, the state of the data model, actors, input actions, system state (sessions), pages (types, fields, and buttons), and output actions.

5.3.3 Results

As stated in the previous section, the comparison of the ATCs generated manually and automatically by the prototype is done based on the structure of the ATC meta-model. Table 5.2 shows the results obtained when comparing the manually and automatically generated ATCs. The columns from left to right in Table 5.2 are: *Test Setup*, *Inputs*, *Expected Outputs*, *FP* - False Positives, *FN* - False Negatives, *Pre* - Precision, *Rec* - Recall, and *F1* - combination of precision and recall. The ATC components are further divided into the following parts, test setup: constraints - subject (*Subj*), data model constraint (*DM-C*), system state constraint (*Sys-C*); inputs: each step - *Actor*, *Action*, *Entity*; and expected outputs (similar to test setup).

False Positives (FP) represent the overall number of data points not labeled in the manual ATCs but generated by the prototype from the user requirements model. False Negatives (FN) represent the overall number of data points labeled in the manual ATCs but are not generated by the prototype from the user requirements model. The number of false positives directly affects the Precision score, as a lower rate of false positives will lead to a higher score. The number of false negatives denotes the number of labels that the model did not manage to predict, which is something we want to minimize. The false negatives affect the Recall score since the lower the number of false negatives is, the higher the Recall score is. The F1 score was calculated as a harmonic mean of Precision and Recall to give a more balanced measure of the

Table 5.2: Results obtained when comparing manually and automatically generated abstract test cases (ATCs). FP - False Positives, FN - False Negatives, Pre - Precision, Rec - Recall, DM-C - Data Model Constraint, Sys-C - System Constraint.

#	Use Cases	Test Setup			Inputs			Expected Outputs			FP	FN	Pre	Rec	F1
		Subj	DM-C	Sys-C	Actor	Action	Entity	Subj	DM-C	Sys-C					
1	PMS-01-Login-Employee	3	2	1	3	3	3	1	1	1	0	3	1.0	0.86	0.92
2	PMS-02-Login-Employer	3	2	1	3	3	3	1	1	1	0	4	1.0	0.82	0.90
3	PMS-21-Logout-Employee	3	2	1	1	1	1	1	NA	1	1	3	0.92	0.79	0.85
4	PMS-21-Logout-Employer	3	2	1	1	1	1	1	NA	1	1	3	0.92	0.79	0.85
5	PMS-05-ApproveTimeSheet	2	2	0	6	6	1	1	1	1	3	4	0.87	0.83	0.85
6	PMS-04-ModifyTimeSheet	2	2	0	17	17	10	1	1	1	24	10	0.68	0.84	0.75
7	PMS-13-AddEmployee	2	2	0	7	5	7	5	5	0	2	14	0.94	0.70	0.80
8	PMS-03-SearchEmployee	3	2	1	1	1	1	5	5	0	3	13	0.86	0.59	0.70
9	PMS-08-SaveTimeSheet	3	2	1	11	11	11	1	1	1	8	11	0.84	0.79	0.82
10	PMS-06-CalculatePay	3	1	1	1	1	1	4	4	0	3	2	0.84	0.89	0.86
11	PMS-09-PayCheck	2	2	0	1	1	1	4	4	0	3	2	0.83	0.88	0.86
12	PMS-07-ChangePwd-Employee	2	2	0	5	5	5	1	1	1	3	15	0.88	0.59	0.71

quality of the approach. We use the following formulas when computing the *Precision*, *Recall*, and *F1 Score* [46].

$$Precision\ (Pre) = TP/(TP + FP) \tag{5.1}$$

$$Recall\ (Rec) = TP/(TP + FN) \tag{5.2}$$

$$F1\ Score = 2 * (Pre * Rec)/(Prec + Rec) \tag{5.3}$$

Row 1 of Table 5.2 shows the data collected when the manually generated ATC for the use case *PMS-02-Login-Employer* is compared to the automatically generated ATC from the prototype. In the test setup there are 3 subject (*Subj*) entries (`Employer`), 2 data model constraints *DM-C* (`has User.User_ID` and `has User.User_PWD`), and one system constraint *Sys-C* (`is on login page`). An example of the ATC is shown on the right side of Figure 5.3. The inputs and expected outputs of the ATC follow the same pattern. The similarity scores for the ATC in Row 1 are as follows. The false positive score is 0 - all data points in the manually generated ATC are the same as the automatically generated ATC. The false negative score is 3 - the manually generated ATC contains additional details at the end of the inputs that identify the fields where the data is entered, e.g., `Employer enters value of User.User_ID in the user_id field`. Using equations 5.1 to 5.3, the Precision, Recall, and F1 score are computed using the false positive and false negative scores. The *NA* in the table states that using the entry in the test cases component is not applicable.

The login use cases in Rows 1 and 2 of Table 5.2 have the highest similarity scores since the transformation process is straightforward, and they only use two attributes from the data model. However, the modify timesheet use case in Row 6 has the lowest Precision score (0.68) between the manually generated ATC and automatically

generated ATC. This score is because there is some repetition in the automatically generated ATC. For example, two entries can be combined that refer to the employer clicking to view the employer information. Row 8, showing the search employer use case has the lowest recall score (0.59) since the false negatives are high relative to the number of entities compared. This score is because the implemented data model structure (database) is different from the data model structure used in the design (ER diagram). We discuss this issue in more detail in the following section.

5.3.4 Discussion

Similarity of automatically and manually generated ATCs. The first research question related to the similarity of ATCs automatically generated, using the MDSE and NLP approach, and manually generated is answered using the results in Table 5.2. In general, the Precision and Recall scores are relatively high, with the lowest score for Precision being 0.68 and the lowest score for Recall being 0.59. These scores can be considered outliers since the other scores for Precision are above 0.83 and for Recall above 0.70. The F1 scores are also relatively high, with the lowest score being 0.70.

The results shown in Table 5.2 are due to the experience gained after generating ATCs using the ER diagram from the student project's design document and realizing that the implemented data model was very different from the design. The first pass using the ER diagram from the design document produced higher numbers of false positives and false negatives, resulting in Precision, Recall, and F1 scores that were lower than the values presented in Table 5.2. For example, the data for the login use case in Row 1 are false positives - 8, false negatives - 5, Precision - 0.62, Recall - 0.72 and the F1 score - 0.67. To obtain the values in Table 5.2 we created an ER diagram for the implemented data model and used it in the prototype.

Limitations of the approach. Several factors affect the generation of accurate ATCs using the approach presented in this paper. Many of these factors are related to consistency between the requirements, design, and implementation of the system. Usually, the semantic gap tends to widen over time between the requirements, design, and implementation. As development progresses, changes are made to the design and not reflected in the requirements. Similarly, changes made during implementation are not reflected in the design. Since testing is performed after the system is implemented, automatically generating test cases from the requirements and design can only be done with a high level of accuracy if there is consistency across the artifacts generated in the requirements, design, and implementation phases.

Since our approach relies on NLP, inherent factors impact the accurate processing of user requirements written in natural language statements. These factors include ambiguity and incompleteness of the natural language statements. In addition, using different terms in the requirements and data model design that refers to the same concept makes it difficult to do exact string matching, thereby forcing the use of alternative approaches. We mitigate these factors by asking the user to perform the match between the substrings or resolve the meaning of a phrase at runtime. The user intervention allows us to create a domain-specific knowledge base that can be used in future string matching or phrase resolutions.

5.4 Summary of Chapter

In this chapter, we presented our solution to SP-II. The approach automatically transforms enhanced user requirements into abstract test cases (ATCs) using model-driven software engineering (MDSE) and natural language processing (NLP). The approach began with constructing an enhanced user requirements (EUR) model in chapter 4. The enhanced user requirement model (EURM) model is created from

the user requirements model (use cases) and a data model (ER diagram) using NLP techniques and an algorithm we developed. To support the transformation from EUR models to abstract test cases (ATCs), we developed meta-models for EURMs and ATCs. A transformation engine was developed to convert EURMs to ATCS using the Eclipse Modeling Framework and the ATLAS Transformation Language.

We evaluated our approach by performing a case study using 12 use cases from a graduate software project, test cases from the graduate project, and four (4) undergraduate software projects. The ATCs generated using the engine were compared to the ATCs manually generated by members of the research team. The comparison involved inspecting the fields of the generated ATCs against the manually created ATCs to determine the precision, recall, and F1 scores. The results were very promising, assuming that the user requirements and data model design were consistent with the implementation of the final system.

CHAPTER 6

GENERATING CONCRETE TEST CASES FROM ABSTRACT TEST CASES

In this chapter, we describe the process of generating concrete test cases from abstract test cases (ATCs). It involves taking high-level, general test scenarios and breaking them down into specific, detailed test steps that can be executed to validate a software application. The process typically includes identifying specific inputs, expected outputs, and conditions for each test case and documenting them in a way that can be easily understood and executed by testers or an automated testing tool. Section 6.1 describes the tool used to generate concrete test cases from user requirements, with an emphasis on the part of the tool that generates concrete test cases from ATCs. Section 6.2 describes a case study on how the various inputs are brought together to generate concrete test cases from user requirements. This chapter presents a solution to *SP-III:* How can concrete test cases be generated given abstract test cases and an instance of a data model derived from user requirements?

6.1 UR2TC Generation Tool

In this section, we describe the high-level architecture of the tool that generates concrete test cases from user requirements written in natural language. We refer to the tool as `UR2TC-GenTool`. Our description will focus more on the last component of the `UR2TC-GenTool` that focuses on generating concrete test cases from ATCs.

6.1.1 Architecture

Figure 6.1 shows the high-level architecture of `UR2TC-GenTool`, which is a composition of the previous tools described in Chapters 4 and 5. Section 4.2 describes the tool that preprocesses user requirements by performing linguistic analysis of the user

requirements' text and checking the conformity of user requirements against the user requirements meta-model. The high-level architecture for the user requirements validation tool is shown in Figure 4.4 and capture on the left side of Figure 6.1. Section 5.2 describes the transformation engine that (a) converts user requirement models into EURMs and (b) transforms EURMs into ATCs that conform to the ATC meta-model. The transformation between EURMs and ATCs is made possible by defining ATL rules and helper functions at the EURM and ATC meta-model level. Figure 5.4 shows the high-level architecture of the transformation engine. The components of the transformation engines are shown in the middle of the Figure 6.1. All the components of `UR2TC-GenTool` use a set of third-party components in the package labeled `Libraries` and a common data store labeled `Repository`.

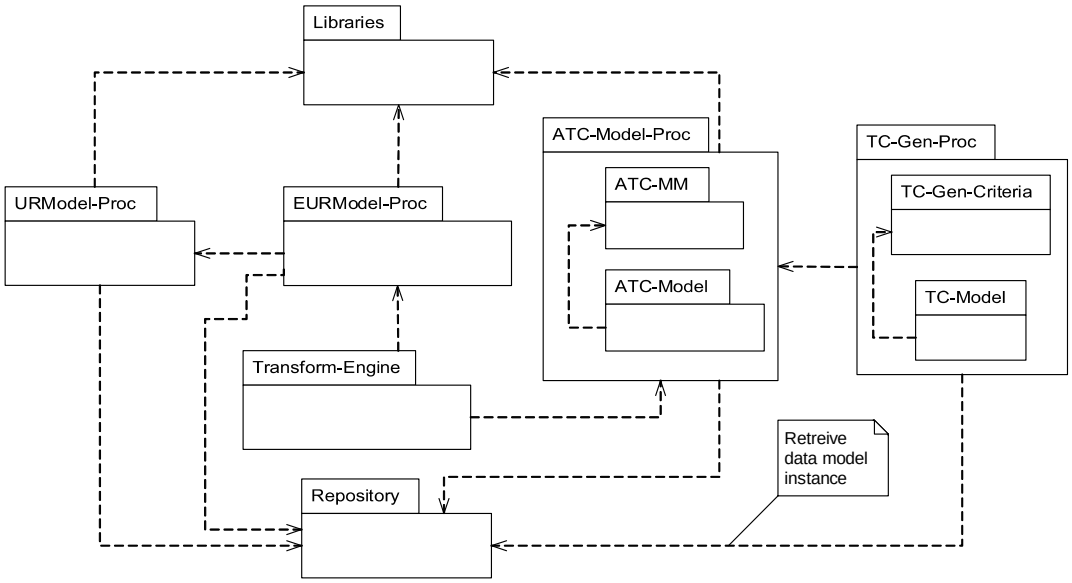

Figure 6.1: Architecture of tool to preprocess requirements models prior to transformation.

The right side of Figure 6.1 shows the last package that was added to complete `UR2TC-GenTool` described in this section. The new package added is `TC-Gen-proc`,

and the updated package is `Repository`. Following is a brief description of these packages.

- `TC-Gen- Criteria` - generates multiple concrete test cases from an abstract test case by accessing an instance of the application's data model in the `Repository`. Using the entities and attributes stated in the ATCs, the data model is queried to select actual data to generate concrete test cases.
- `TC-Model`- stores the concrete test cases generated using the test generation criteria defined in the package `TC-Gen- Criteria`.
- `Repository`- store the data model instance. This model conforms to the data model used to generate the EURM in Section 4.4.2.

6.1.2 Algorithm

Sunny day testing and rainy day testing are two types of software testing that focus on different scenarios see Algorithms 6.1 and 6.2.

After the transformation to abstract test cases, they are then processed using either of Algorithms 6.1 or 6.2 to generate concrete test case model (CTCM). The input for both the algorithms is ATCM - *Abstract Test Case Model* and Repos - *includes Data model Instance, Data pool, Test generation rules.* The CTCM algorithm starts with line 1 gen_CTC (ATCM, *ref* Repos, Tcriterion, Ttype). If the CTC we are generating is a sunny day test case then Tcriterion is SUNNY in line 4 and if its rainy day test case then Tcriterion is RAINY of the respective algorithms.

There are 4 types of Ttypes as rules in this algorithm *READ*, *CREATE*, *UPDATE* and *DELETE*. For rules in line 4, 8, 12 and 17 we collect List.fields(<attr, values>) by executing query in repository of data model of instance for List.fields(<attr, NULL>)). For Example *UPDATE*, list of fields can be both from data model instance or data pool we generated. For Algorithm 6.2, we perform different mutations

on the list of fields depending on the Ttype. As an example, for a *READ* we mutate a field in the list by changing the string for the field shown in test case in Figure 6.4(b). After matching the rules, finally a CTCM is generated in line 22 from ATCM, List.fields(<attr, values>), Repos.rules and optype. We plan to fully define the mutation operator in future work.

Algorithm 6.1 Algorithm showing the generation of sunny CTC generation

```
1: gen_CTC (ATCM, ref Repos, Tcriterion, Ttype)
   /*Input: ATCM - abstract test case model;
     Repos - includes data model instance (DMI), data pool, test generation rules
   Output: CTCM- concrete test case model; */
2: CTCM ← createCTCM()
3: if (Tcriterion == SUNNY) then
4:    if (Ttype == READ) then
5:       List.fields(<attr, values>) ← query(Repos.DMI, List.fields(<attr, NULL>))
6:       optype ← sunny_read
7:    end if
8:    if (Ttype == CREATE) then
9:       List.fields(<attr, values>) ← query(Repos.datapool, List.fields(<attr, NULL>))
10:      optype ← sunny_create
11:   end if
12:   if (Ttype == UPDATE) then
13:      List.fields(<attr, values>) ← query(Repos.DMI, List.fields(<attr, NULL>))
14:      List.fields(<attr, values>) ← query(Repos.datapool, List.fields(<attr, NULL>))
15:      optype ← sunny_update
16:   end if
17:   if (Ttype == DELETE) then
18:      List.fields(<attr, values>) ← query(Repos.DMI, List.fields(<attr, NULL>))
19:      optype ← sunny_delete
20:   end if
21: end if
22: CTCM ← generate(ATCM, List.fields(<attr, values>), Repos.rules, optype)
23: return CTCM
```

Algorithm 6.2 Algorithm showing the generation of rainy CTC generation

```
1: gen_CTC (ATCM, ref Repos, Tcriterion, Ttype)
   /*Input: ATCM - abstract test Case Model;
     Repos - includes data model instance (DMI), data pool, test generation rules
   Output: CTCM - concrete test case model; */
2: CTCM ← createCTCM()
3: if (Tcriterion == RAINY) then
4:   if (Ttype == READ) then
5:     List.fields(<attr, values>) ← query(Repos.datapool, List.fields(<attr, NULL>))
6:     mutateR(List.fields) /* Modifies a field in list */
7:     optype ← rainy_read
8:   end if
9:   if (Ttype == CREATE) then
10:    List.fields(<attr, values>) ← query(Repos.datapool, List.fields(<attr, NULL>))
11:    mutateC(List.fields) /* Nullifies a required field in list */
12:    optype ← rainy_create
13:  end if
14:  if (Ttype == UPDATE) then
15:    List.fields(<attr, values>) ← query(Repos.DMI, List.fields(<attr, NULL>))
16:    List.fields(<attr, values>) ← query(Repos.datapool, List.fields(<attr, NULL>))
17:    mutateU(List.fields) /* Nullifies a field in list */
18:    optype ← rainy_update
19:  end if
20:  if (Ttype == DELETE) then
21:    List.fields(<attr, values>) ← query(Repos.DMI, List.fields(<attr, NULL>))
22:    mutateD(List.fields) /* Modifies a primary key in list */
23:    optype ← rainy_delete
24:  end if
25: end if
26: CTCM ← generate(ATCM, List.fields(<attr, values>), Repos.rules, optype)
27: return CTCM
```

6.1.3 Implementation

Before concrete test cases can be created, there need to be one or more ATCs generated by the `ATC-Model-Proc` and a data model instance stored in the repository conforming to the data model used to create the EURM in Section 4.4.2. The procedures to develop concrete test cases are defined in the `TC-Gen-proc` package. Based on the test data generation criteria defined in the package `TC-Gen-Criteria`, concrete test cases are generated from ATCs. In our implementation, were have defined the procedure that generates test cases based on *sunny day* scenarios, referred to as `TCSunny` and *rainy day* scenarios, referred to as `TCRainy` [48].

A *sunny day* scenario refers to an instance of a use case that adheres to the preconditions and follows the steps in the description as stated without deviation. On the other hand, the *rainy day* scenario can have variations in the preconditions or the description sections of the use case. These variations are usually defined as alternatives or exceptions in the user requirement [21].

The process to generate concrete test cases from ATCs is as follows. References are made to the packages in 6.1 to concretize control flow.

1. The `TC-Model` request a ATC from the repository.

2. Using the ATC, a test case id and purpose are generated for the concrete test case from the ATC's *Abstract Test Case ID* and *Purpose* sections, respectively.

3. The `TC-Model` invokes the `TC-Gen-Criteria` to determine which test generation approach will be used. In our case, we use the sunny day/rainy day scenario approach.

4. Based on the test generation approach selected, each entry in the ATC *Test Setup* section is analyzed in the context of the data contained in the data model instance. For example, if we use the sunny day scenario approach for test data

generation, then the data fields stated in the test setup must exist and have actual data in the data instance model. In addition to the data in the data model instance, the test setup can specify the state the system is in or the page currently displayed by the system.

5. The entries in the *Inputs* section of the ATC are processed as follows:

   (a) If an entry refers to a specific field in the data model, then depending on the test generation approach used, data is copied from the data model instance, and the appropriate change is applied to that piece of data. In our case, using the sunny day scenario approach, the data fields in the ATC are replaced with specific data. Using the rainy day approach, at least one of the input fields must have different data than the data stored in the data model instance.

   (b) If the entry is associated with a specific action related to a widget on a particular page, then the appropriate action is performed again depending on the type of test generation approach.

6. The *Expected Output* entry in the ATC is copied to the expected output section of the concrete test case. This is a limitation of our approach since we do not create any behavioral models during our approach to determine the output given specific inputs.

6.2 Case Study

In this section, we present a case study that shows the artifacts from the PMS system to are used to generate concrete test cases. The artifacts shown are two use cases, the entity-relation diagram representing the data model used in the implementation of the PMS application and the associated data model instances. The entity-relation

diagram shown in this case study is slightly different from the diagram shown in Figure 3.3, which is taken from the design document for PMS.

Although we do not perform an extensive validation of our approach in generating concrete test cases for the user requirements, the case studies presented in Chapters 4 and 5 provide confidence in generating ATCs from user requirements. Based on the examples provided and our experience using the 6.1 we discuss the limitations of our approach.

6.2.1 Setup

The artifacts used in the case study are required to successfully generate the test cases from user requirements written as use cases. The artifacts include:

- The use case *PMS-02-Login-Employer* - Employer enters their username and password to log into the system. This use case is the one used as the illustrative example throughout the dissertation, see Figure 3.2.
- The use case *PMS-13-AddEmployee* - Employer adds a new Employee to the system by visiting the Manage tab, see Figure 6.2
- The entity relation diagram showing the application-specific data model for the PMS application, see Figure 6.3.
- The Employer table for the data model instance for the PMS data model, see Table 6.1.
- The Employee table for the data model instance for the PMS data model, see Table 6.2.

The structure of the both the uses cases in Figures 3.2 and 6.2 is similar since they are both instances of the UR meta-model in Section 4.1. The entity- relation

Use Case ID: PMS-13-AddEmployee
Goal:
The Employer adds a new Employee to the system by visiting the Manage tab
Actor: Employer
Precondition:
1. The Employer is logged in and on the homepage
2. The manage tab is active

Description:
1. The use case begins when the Employer clicks on the "manage" option
2. The system brings up the option to manage Employee
3. The Employer selects Employee in the option
4. The Employer then selects "Add Employee" option
5. The Employer then fills in the required fields: employee id, first name, last name, gender, date of birth, job role, phone number, email address, address, bank account number, bank name, join date

Postcondition:
1. The system stores the new Employee information
2. The number of Employees in the system increase by one

Figure 6.2: Example of the add employee use case by the employer for the Payroll Management System (PMS)

diagram contains 7 entities, including `Employer`, `Employer`, `Emp_TS` - time sheet for the employees, `PayModel` - whether the employee's pay is the normal salary or extra pay, `Salaries` - details of the employee salaries, `Save_TS` - save time sheet after validation, and `Users` - security information fro the employee account. A review of the design document and implementation for the PMS system shows a lack of consistency between the entity-relation diagram in the design document and the actual implementation of PMS.

Table 6.1 shows the data model instance for a Employer consisting of the attributes `username` and `password`. The table has two entries for *user1* and *user2*. Table 6.2 contains the data model instance for and Employee and has 12 fields, the field names are highlighted in bold. The table has 3 sections each with four fields. Table 6.2 will

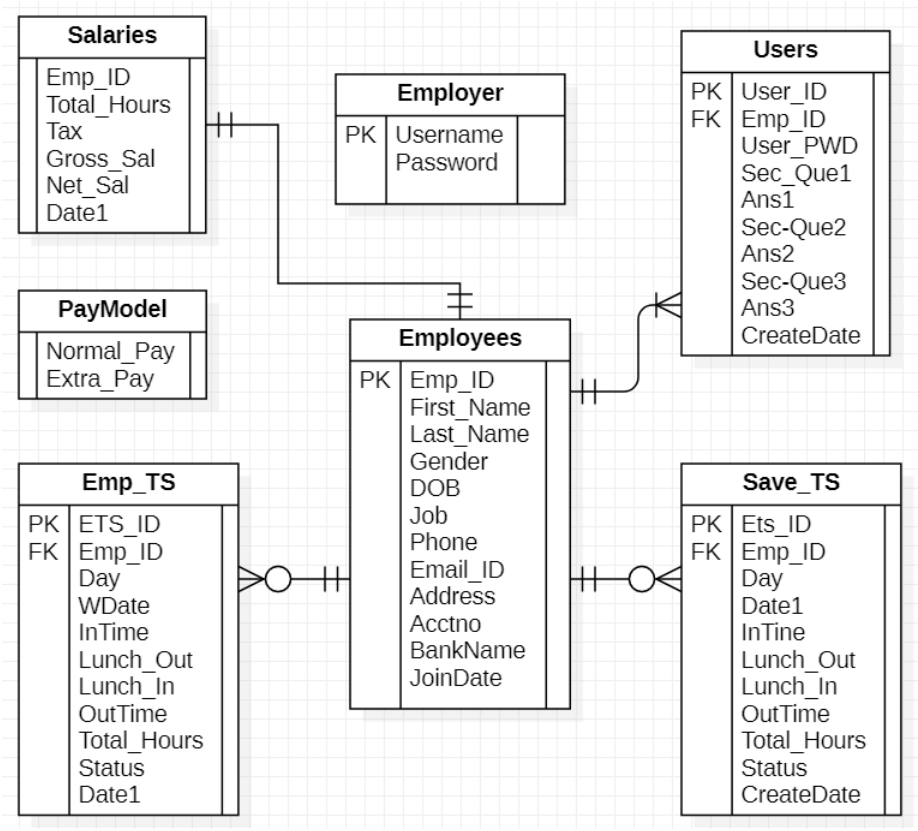

Figure 6.3: Entity-relation diagram (ERD) used for PMS in the student project implementation.

be populated with data after the execution of the test cases described in the next section.

6.2.2 Results

In this section, we describe the results of the generated sunny day and rainy day test cases for the employer login use case *PMS-02-Login-Employer*, Figure 3.2, and the add employee use case *PMS-13-AddEmployee*, Figure 6.2. The results for the sunny day and rainy day test cases generated by the `UR2TC-GenTool` for employer login and add employee are shown in Figures 6.4 and 6.5, respectively.

We explain in detail the results of the test cases shown in Figure 6.4 for the use case *PMS-02-Login-Employer* in the following enumerated list.

Table 6.1: Data model instance of table Employer

username	**password**
user1	pass1234
user2	pass4321

Table 6.2: Data model instance of table Employees

Emp_ID	**First_Name**	**Last_Name**	**Gender**
DOB	**Job**	**Phone**	**Email_ID**
Address	**Acct_No**	**Bank_Name**	**Join_Date**

Test Case ID: PMS_02_ login-Employer-TC001-Sunny
Purpose: Employer enters his/her username and password to login to the system
Test Setup:
Employer has username user1
Employer has password pass1234
Employer is login page
Input:
employer enters username user1
employer enters password pass1234
employer clicks signin button
Expected Output:
pms system redirected Employer homepage

(a) PMS_02 Sunny Day Test Case

Test Case ID: PMS_02_ login-Employer-TC002-Rainy
Purpose: Employer enters his/her username and password to login to the system
Test Setup:
Employer has username user1
Employer has password pass1234
Employer is login page
Input:
employer enters username *user1rainy*
employer enters password *pass1234rainy*
employer clicks signin button
Expected Output:
pms system *not* redirected Employer homepage

(b) PMS_02 Rainy Day Test Case

Figure 6.4: Test cases generated using the data model instance for use case PMS_02_login-Employer for PMS. (a) Sunny day test case. (b) Rainy day test case.

- Figure 6.4 shows the test cases generated using the data model instance in Tables 6.1 and 6.2. The left side of Figure 6.4 shows a sunny day test case (a), and the right side is a rainy day test case (b). These test cases are used to check the correctness of (a) a sunny day scenario and (b) a rainy day scenario for the use case *PMS-02-Login-Employer*. All inputs and outputs are within normal ranges for each data type. The type of test case, sunny or rainy, is determined by a combination of the test setup and the input values.

- The first two sections of the test case `Test Case ID:` and `Purpose` are easily generated from the use case. In this example, the test case ids for the sunny day and rainy day test cases are *PMS_02_login-Employer-TC001-Sunny* and *PMS_02_login-Employer-TC002-Rainy*, respectively. The purpose for both test cases is the same. Not that the purpose for the rainy day test case can be refined to be more specific.

- The test setup for both test cases ensures that an `Employer` account exists in the system and that the login credentials `username` = *user1* and `password` = *pass1234* are associated with an `Employer`, see Figure 6.4, section `Test Setup`. The values for the `username` and `password` are selected from the data model instance shown in Table 6.1.

- The input section for the sunny test case requires the input for the `username` field is *user1* and for the `password` field is *pass1234*. This input is generated by selecting appropriate values from the data model instance, see Table 6.1 and must match the test setup values. For the rainy day test case, the input is generated by replacing one or both input fields with data not in Table 6.1. In this scenario, the input data selected is *user1rainy* for the `username` field and *pass1234rainy* for the `password` field. The last action in the input section of the test case is the same for both test cases, e.g., *employer clicks signin button*.

- The last section of the test cases is the expected output. The main difference in the expected output between the sunny day test case and the rainy day test case is that the rainy day test case negates the expected output for the sunny test cases. For example, *pms system not redirected Employer homepage*.

We explain in detail the results of the test cases shown in Figure 6.5 for the use case *PMS-13-AddEmployee* in the following enumerated list.

- Figure 6.5 shows the test cases generated using the data model instance in Tables 6.1 and 6.2. The left side of Figure 6.5 shows the sunny day test case, and the right side the rainy day test case for the use case *PMS-13-AddEmployee*.

- The test case id and purpose sections are similar to those for the previous test case but are specific to the add employee use case.

- The test setup section ensures that the `Employer` account exists in the system and that the login credentials are correct. The credentials are `username` = *user1* and `password` = *pass1234* and are available for the `Employer` to access the `PMS system` to add an employee as shown in the figure under the section `Test Setup`. It should be noted that the precondition in use case *PMS-13-AddEmployee* is not complete since an employee with the same employee id should not be already in the system.

- The inputs for the sunny day test case shown on the left of Figure 6.5 are as follows: `emp_Eid` = *1*, `first_name` = *Adam*, `last_name` = *Sandler*, `gender` = *on*, `dob` = *1901−01−01*, `job` = *Movie Star*, `phone` = *0*, `email_id` = *adam.sandler @email.com*, `address` = *2121 SW 12TH ST*, `accno` = *1*, `bankname` = *Bank of America*, and `joindate` = *2020−01−14*. The final entry in the `Input` section is *employer clicks add button*. The inputs selected for the sunny day test cases can be identified from a data pool of application-specific data for each field required.

  The input for the rainy day test case is similar to the sunny day test case, except the `emp_Eid` field is left blank. We assume that no input field may be left blank since all fields are required.

- The expected output for the sunny day test case is that the `PMS system stores` the new employee information into the system as stated in the `Expected Output` section of the test case. The system's state has changed after entering the input

Test Case ID: PMS_13_AddEmployee-TC001-Sunny
Purpose: The Employer adds a new Employee to the system by visiting the Manage tab
Test Setup:
employer has username user1
employer has password pass1234
employer is logged in
Input:
employer enters emp_id: 1
employer enters first_name: Adam
employer enters last_name: Sandler
employer enters gender: on
employer enters dob: 1901-01-01
employer enters job: Movie Star
employer enters phone: 0
Employer enters email_id: adam.sandler@email.com
employer enters address: 2121 SW 12TH ST
employer enters accno: 1
employer enters bankname: Bank of America
employer enters joindate: 2020-01-14
employer clicks add button
Expected Output:
pms system store new employee information

(a) PMS_13 Sunny Day Test Case

Test Case ID: PMS_13_AddEmployee-TC002-Rainy
Purpose: The Employer adds a new Employee to the system by visiting the Manage tab
Test Setup:
employer has username user1
employer has password pass1234
employer is logged in
Input:
employer enters emp_id:
employer enters first_name: Adam
employer enters last_name: Sandler
employer enters gender: on
employer enters dob: 1901-01-01
employer enters job: Movie Star
employer enters phone: 0
Employer enters email_id: adam.sandler@email.com
employer enters address: 2121 SW 12TH ST
employer enters accno: 1
employer enters bankname: Bank of America
employer enters joindate: 2020-01-14
employer clicks add button
ExpectedOutput:
pms system *not* store new employee information

(b) PMS_13 Rainy Day Test Case

Figure 6.5: Test cases generated using the data model instance for use case PMS_13_AddEmployee for PMS. (a) Sunny day test case. (b) Rainy day test case.

data, which is reflected in the updated data model instance as shown in Table 6.3. There is no change to the data model instance for the rainy day test case, as stated in the expected output section of the rainy day test case - *pms system not store new employee information*.

6.2.3 Discussion

The overall approach used to validate the end product of the process described in Chapter 3 and shown in Figure 3.1, although limited, demonstrates the feasibility of the approach and provides insight into the challenges of generating test cases from user requirements written in natural language. Using the data model throughout the

Table 6.3: Data model instance of table Employees

Emp_ID	**First_Name**	**Last_Name**	**Gender**
1	Adam	Sandler	on
DOB	**Job**	**Phone**	**Email_ID**
1901−01−01	Movie Star	0	adam.sandler@email.com
Address	**Acct_No**	**Bank_Name**	**Join_Date**
2121 SW 12TH ST	1	Bank of America	2020−01−14

transformation process enhances the ability to easily generate test cases with actual data from the data model instance. The validation approach presented in this chapter could have been more robust by comparing our approach to other approaches in the literature [6]. The data set used for validating the case study described in Chapter 5 can be easily extended and compared to other techniques presented in the literature and described in Chapter 2. Recall that we reverse-engineered the concrete test cases in Chapter 5 to obtain the abstract test cases.

Based on our approach to generating test cases from the data model instance, generating additional test cases, specifically rainy day test cases, is relatively easy. Our approach to generating the rainy day test cases is to manipulate the input by entering values not in the data model, see Figure 6.4, or omitting values for required data entry fields, see Figure 6.5. Although not presented in the case study in this chapter, the overall quality of the test cases generated can be evaluated using techniques, such as mutation testing technique [4], which is out of the scope of this dissertation. The overall effort of generating the test cases from user requirements can also be evaluated using appropriate techniques [29, 17].

Limitations of the approach. Consistency is a key factor for creating test cases based on use cases. Use cases affects testability. Traceability across the project is important when it comes to names or abbreviations used in domain specific application across all use cases and also data models. Another critical issue recognized that natural language requirements such as ambiguity and incompleteness of natural lan-

guage statements significantly impact software testing. Names used across the system for like database entity and attribute names should either be abbreviations or sub strings of the words used in use cases. As a general practice, given a narrative description of the database requirements, the nouns appearing in the narrative tend to give rise to entity type names, and the verbs tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types. Finding and modifying the names is needed by re-engineering/updating use cases through out the development process. This will attain efficiency and accuracy to the system.

Threats to Validity. The main threat to validity in our study relates to semantic understanding of the system. To deal with this threat, we used a knowledge base and put rules in place for naming ER design. When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. We used a small set of knowledge base keywords for the domain, we tried update these keywords from user as we explore new use cases.

6.3 Summary of Chapter

In this chapter, we presented our solution to SP-III. The approach began with process of generating concrete test cases from abstract test cases (ATCs). It involves taking high-level, general test scenarios and breaking them down into specific, detailed test steps that can be executed to validate a software application. The process typically includes identifying specific inputs, expected outputs, and conditions for each test case and documenting them in a way that can be easily understood and executed by testers or an automated testing tool. Section 6.1 describes the tool used to generate concrete test cases from user requirements, with an emphasis on the part of the tool

that generates concrete test cases from ATCs. Section 6.2 described a case study on how the various inputs are brought together to generate concrete test cases of both sunny and rainy from user requirements.

CHAPTER 7

CONCLUSION

In this chapter, we present a summary of the dissertation and future work in the area of generating test cases from user requirements. The summary of our research identifies the sub-problems investigated and recaps how the work presented in this dissertation addresses them. The future work section describes potential topics to be investigated based on the solutions to the sub-problems presented. In addition, it shows how the tool built for solving the three sub-problems in this dissertation can be extended. In Section 7.1 the summary of my research is presented, and Section 7.2 describes the future work indicating the direction in which the field may develop and advance.

7.1 Summary of Research

This section summarizes the research presented in this dissertation and highlights the solutions to the research problem presented in Chapter 1. An overview of the solution to the research problem is presented in Chapter 3. The sub-problems investigated are: (I) How can user requirements written as structured Natural Language along with the associated data model be represented in a semi-formal manner to support testing? (II) How can abstract test cases be generated given the semi-formal representation of a user requirements model? and (III) How can concrete test cases be generated given abstract test cases and an instance of a data model derived from user requirements?

The solution to sub-problem-I is presented in Chapter 4. The solution includes creating an enhanced user requirements model (EURM) from commonly used user requirement models (URMs), e.g., use cases and user stories. The first stage is creating a meta-model for URMs and validating that the meta-model is correct. The next stage

is extending the meta-model for URMs to include elements of the data model for a specific application and results of NLP performed on the text of the user requirements. As a result, a meta-model for EURMs was created, along with the algorithms to support the process of creating EURMs from URMs.

To validate the correctness of the user requirements meta-model, we developed a tool that takes the URM as input and generates feedback to the user in terms of warnings and errors related mainly to the syntax of the URM. A case study is presented that validates the user requirements for 50 use cases from undergraduate student projects and 50 use stories from an industrial application. Overall the results were positive, showing that the meta-model is very close to being correct. The meta-model for EURMs is validated using experiments performed in Chapter 5 when transforming EURMs to abstract test cases.

The solution to the sub-problem-II is presented in Chapter 5. Using the approach that began with constructing an EURM in chapter 4, i.e., creating EURMs from URMs (use cases) that require a data model (ER Diagram), appropriate NLP techniques, and an algorithm we developed. To support the transformation from EURMs to abstract test cases (ATCs), we developed a meta-model for ATCs. A transformation engine was developed to convert EURMs to ATCS using the Eclipse Modeling Framework [19] and the ATLAS Transformation Language (ATL) [33]. The transformation engine includes rules defined using ATL between the EURM and ATC meta-models, as shown in the solution overview in Chapter 3.

We evaluated the transformation from EURMS to ATCs by performing a case study using 12 use cases from a graduate software project, test cases from the graduate project, and four (4) undergraduate software projects. The ATCs generated using the transformation engine were compared to the ATCs manually generated by members of the research team. The comparison involved inspecting the fields of the generated

ATCs against the manually created ATCs to determine the precision, recall, and F1 scores [46]. The results were very promising, assuming that the user requirements and data model design were consistent with the implementation of the final system.

The solution to the sub-problem-III is presented in Chapter 6. This solution is centered on the process of generating concrete test cases from abstract test cases (ATCs). The process involves taking high-level, general test scenarios and breaking them down into specific, detailed test steps that can be executed to validate a software application. The process typically includes identifying specific inputs, expected outputs, and conditions for each test case and documenting them in a way that can be easily understood and executed by testers or an automated testing tool. The key to generating concrete test cases is having a data model instance that is consistent with the structure of the data model used to build the EURMs. We describe the UR2TC tool used to generate concrete test cases from user requirements, emphasizing the part of the tool that generates concrete test cases from ATCs. A case study is presented that describes how the various inputs are brought together to generate concrete test cases for both sunny and rainy scenarios from the user requirements.

7.2 Future Work

The research framework for generating test cases from user requirements is presented in Chapter 3 of this dissertation and can be the basis for extending the current research and investigating several interesting and exciting projects. The immediate research projects include extending the case study presented in Chapter 6 to make a comparison of existing test generation techniques using common data sets or the data sets used in the related work. Such a project would compare the quality of the test cases generated in our work and the other works cited in the literature. We would also like to compare the effort required to generate test cases using our approach and

other works cited in the literature. In addition, we plan to extend our work to include test case generation for user stories and for non-functional and security requirements for use cases.

The longer-term projects include: (1) Investigating how ML algorithms can be trained to automatically generate test cases based on user requirements. (2) Using future NLP technology to analyze user requirements. Future NLP techniques are likely to become even more sophisticated and capable of accurately understanding and interpreting human language. This will enable the automation of the requirements gathering and analysis process, where NLP algorithms can extract requirements from unstructured text compared to the structured text used in this research and convert them into a structured format that can be used to drive the software development process. (3) Using other Artificial Intelligence (AI) techniques and tools to perform requirements analysis to extract and process user requirements from natural language specifications.

The future of Natural Language Processing (NLP) and Model-Driven Software Engineering (MDSE) based automation from requirements to test cases is promising and holds a lot of potential. NLP and MDSE are two areas of technology that are advancing rapidly and can be leveraged to automate the software development process from requirements to test cases.

Combining the strengths of AI, NLP, and MDSE has the potential to revolutionize the software development process, making it possible to fully automate many phases of the development lifecycle. Using AI and NLP to automatically generate test cases based on non-functional requirements, such as performance, security, and scalability, would be feasible using our meta-models. The advances in AI, NLP, and MDSE could lead to an even faster and more efficient software development process, improved

software quality, and a reduction in the cost and time required to develop and maintain software.

Overall, the future of AI, NLP, and MDSE in the automation from requirements to test cases is very exciting, and we can expect to see continued advancements in these areas in the coming years.

BIBLIOGRAPHY

[1] Imran Ahsan, Wasi Haider Butt, Mudassar Adeel Ahmed, and Muhammad Waseem Anwar. A comprehensive investigation of natural language processing techniques and tools to generate automated test cases. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago, Tariq M. King, and Peter J. Clarke. Towards transforming user requirements to test cases using mde and nlp. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 350–355, 2019.

[3] Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago, Tariq M. King, and Peter J. Clarke. Generating abstract test cases from user requirements using MDSE and NLP. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security*, volume 2, 2022.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[5] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[7] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013.

[8] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: Approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, page 250–260, New York, NY, USA, 2016. Association for Computing Machinery.

[9] C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, Sep. 2003.

[10] V. Bajpai and R. P. Gorthi. On non-functional requirements: A survey. In *2012 IEEE Students' Conference on Electrical, Electronics and Computer Science*, pages 1–4, March 2012.

[11] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012.

[12] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Nat2testscr: Test case generation from natural language requirements based on scr specifications. *Science of Computer Programming*, 95:275–297, 2014. Special Section: ACM SAC-SVT 2013 + Bytecode 2013.

[13] Samira Si-said Cherfi, Jacky Akoka, and Isabelle Comyn-Wattiau. Use case modeling and refinement: A quality-based approach. In David W. Embley, Antoni Olivé, and Sudha Ram, editors, *Conceptual Modeling - ER 2006*, pages 84–97, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[14] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[15] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[16] Dennis Cohn-Muroy and José Antonio Pow-Sang. Can user stories and use cases be used in combination in a same project? a systematic review. In Jezreel Mejia, Mirna Munoz, Álvaro Rocha, and Jose Calvo-Manzano, editors, *Trends and Applications in Software Engineering*, pages 15–24, Cham, 2016. Springer International Publishing.

[17] Érika Regina Campos de Almeida, Bruno Teixeira de Abreu, and Regina Moraes. An alternative approach to test effort estimation based on use cases. In *2009 International Conference on Software Testing Verification and Validation*, pages 279–288, 2009.

[18] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. *Model Transformations*, pages 91–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[19] Eclipse Foundation Inc. Eclipse Modeling Framework. `https://www.eclipse.org/modeling/emf/`, 2019. [Online; accessed 10-Jan-2019].

[20] Robson Do Nascimento Fidalgo, Elvis Maranhão De Souza, Sergio España, Jaelson Brelaz De Castro, and Oscar Pastor. Eermm: A metamodel for the enhanced entity-relationship model. In *Proceedings of the 31st International Conference on Conceptual Modeling*, ER'12, page 515–524, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Donald Firesmith. Generating complete, unambiguous, and verifiable requirements from stories, scenarios, and use cases. *J. Object Technol.*, 3(10):27–40, 2004.

[22] Jannik Fischbach, Andreas Vogelsang, Dominik Spies, Andreas Wehrle, Maximilian Junker, and Dietmar Freudenstein. Specmate: Automated creation of test cases from acceptance criteria. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 321–331, 2020.

[23] Vahid Garousi, Sara Bauer, and Michael Felderer. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology*, 126:1–20, 2020. Paper no. 106321.

[24] Ralph Gerbig, Juan Cadavid, and Adolfo S. The OCLinEcore Language. `https://wiki.eclipse.org/OCL/OCLinEcore`, 2019. [Online; accessed 10-Jan-2019].

[25] Robin Gröpler, Viju Sudhi, Emilio José Calleja García, and Andre Bergmann. NLP-based requirements formalization for automatic test case generation. In *29th International Workshop on Concurrency, Specification and Programming (CS&P 2021)*, pages 31–45. CEUR, 2021.

[26] J.J. Gutiérrez, M.J. Escalona, and M. Mejías. A model-driven approach for functional test case generation. *Journal of Systems and Software*, 109:214–228, 2015.

[27] Torben Mejlvang Hangensen and Bent Bruun Kristensen. Consistency in software system development: Framework, model, techniques & tools. *SIGSOFT Softw. Eng. Notes*, 17(5):58–67, November 1992.

[28] Chu Hue, Duc-Hanh Dang, Nguyen Binh, and Hoang Truong. Usltg: Test case automatic generation by transforming use cases. *International Journal of Software Engineering and Knowledge Engineering*, 29:1313–1345, 09 2019.

[29] IBM. Engineering lifecycle management solution 7.0.3. `https://www.ibm.com/docs/en/elm/7.0.3`, 2022. Accessed: 2023-02-16.

[30] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, Cambridge, MA, USA, 2012. Revised Edition.

[31] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[32] Kunxiang Jin and Kevin Lano. Generation of test cases from uml diagrams - a systematic literature review. In *14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] Krishna Kalyanathaya, Akila D., and Suseendran G. A fuzzy approach to approximate string matching for text retrieval in nlp. *Journal of Computational Information Systems*, 15:26–32, 05 2019.

[35] Mohamad Kassab, Joanna F. DeFranco, and Phillip A. Laplante. Software testing: The state of the practice. *IEEE Software*, 34(5):46–52, 2017.

[36] Divya Kumar and Krishn Mishra. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, 79:8–15, 12 2016.

[37] Beatriz Pérez Lamancha, Macario Polo, Danilo Caivano, Mario Piattini, and Giuseppe Visaggio. Automated generation of test oracles using a model-driven approach. *Information and Software Technology*, 55(2):301–319, 2013. Special Section: Component-Based Software Engineering (CBSE), 2011.

[38] Elizabeth D. Liddy. Natural language processing. In *Encyclopedia of Library and Information Science*. Marcel Decker, Inc., New York, NY, 2001.

[39] Akalanka Mailewa, Jayantha Herath, and Susantha Herath. A survey of effective and efficient software testing. In *The Midwest Instruction and Computing Symposium.(MICS), Grand Forks, ND*, 2015.

[40] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Prismatic Inc, Steven J. Bethard, and David Mcclosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[41] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Prismatic Inc, Steven J. Bethard, and David Mcclosky. The stanford corenlp natural lan-

guage processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[42] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[43] Object Management Group. Object Constraint Language. `https://www.omg.org/spec/OCL/`, 2014. [Online; accessed 07-Feb-2019].

[44] Object Management Group. XML Metadata Interchange Specification. `https://www.omg.org/spec/XMI/`, 2019. [Online; accessed 07-Feb-2019].

[45] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[46] David Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation (tech. rep.). Technical Report SIE-07-001, School of Informatics and Engineering Flinders University, Adelaide, Australia, December 2007.

[47] Muthu Ramachandran. Requirements-driven software test: A process-oriented approach. *SIGSOFT Softw. Eng. Notes*, 21(4):66–70, July 1996.

[48] Doug Rosenberg, Barry Boehm, Matt Stephens, Charles Suscheck, Shobha Rani Dhalipathi, and Bo Wang. *Test Early, Test Often*, pages 107–130. Springer International Publishing, Cham, 2020.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[50] Dionny Santiago, Justin Phillips, Patrick Alt, Brian Muras, Tariq M. King, and Peter J. Clarke. Machine learning and constraint solving for automated form testing. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 217–227, 2019.

[51] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. Automating test automation. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 881–891, 2012.

[52] C. Wang, F. Pastore, A. Goknil, and L. C. Briand. Automatic generation of acceptance test cases from use case specifications: An nlp-based approach. *IEEE Transactions on Software Engineering*, 48(02):585–616, feb 2022.

[53] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 385–396, New York, NY, USA, 2015. ACM.

[54] Xinyu Wang, Liping Zhao, Ye Wang, and Jie Sun. The role of requirements engineering practices in agile development: An empirical study. In *APRES*, 2014.

Appendix

APPENDICIES

```
<?xml version="1.0" encoding="UTF-8"?>
<UseCase xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://org/eclipse/example/Require"
    xsi:schemaLocation="http://org/eclipse/example/Require
    ../../src/main/java/UserRequirements/Requirements.ecore"
    name="PMS_02_ login-Employee"
    description="Employee enters his/her username and password to
    login to the system">
  <actornameusecase>
    <ActorName>Employee</ActorName>
  </actornameusecase>
  <preusecase>
    <preexpr>The Employee has an account</preexpr>
    <preexpr>The Employee has the correct credentials</preexpr>
    <preexpr>The Employee is on the login page</preexpr>
  </preusecase>
  <SECusecase>
    <usecaserightstate>PMS system authenticates username
    and password</usecaserightstate>
    <usecaserightstate>The Employee is redirected to the
    Employee homepage</usecaserightstate>
    <usecaserightstate>The Employee can use PMS system
    functionalities after successful login</usecaserightstate>
  </SECusecase>
  <descusecase>
    <steps>
      <Action>The Employee enters the username in the login page</Action>
      <Action>The Employee enters the password in the login page</Action>
      <Action>The Employee then clicks on the sign_in
      button in the login page</Action>
    </steps>
  </descusecase>
</UseCase>
```

Figure A1: XMI for Use Case Login

Algorithm 7.1 Algorithm showing the generation of an EURM from a user requirements model and data model

```
1: gen_EURM (URM, ref Repos)
   /*Input: URM - User Requirements Model;
     Repos - Repos - repository for NLP entities and
             data model (DM)
   Output: EURM - enhanced user requirements model; */
     Repos - updated repository */
2: eurm ← createEURM()
3: preproc_URM(URM, Repos) /*See line 39*/
4: for each entryType ∈ URM do
5:    if entryType.equals(UC_ID) then
6:       eurm.addUseCase_ID(entry)
7:    end if
8:    if entryType.equals(GOAL) then
9:       eurm.addGoal(entry)
10:   end if
11:   if entryType.equals(ACTOR) then
12:      /*performs fuzzy substring matching with annotated SROs*/
13:      /*if no match or match probability low ask user*/
14:      matchActor = matchingActor(entry, Repos)
15:      eurm.addActor(matchActor, Repos) /*checks DM for aliases*/
16:   end if
17:   if entryType.equals(PRECOND) then
18:      for each entry ∈ precond do
19:         /*matchPrecond is a list containing CEntity1, COperator,*/
20:         /* CEntity2*/
21:         matchPrecond = matchingPrecond(entry, Repos)
22:         /*checks DM for aliases in addPrecond*/
23:         eurm.addPrecond(matchPrecond, Repos)
24:      end for
25:   end if
26:   if entryType.equals(DESCRIPT) then
27:      for each entry ∈ descript do
28:         /*matchDescript is a list containing SSubject, SAction, */
29:         /* SObject, SDest*/
30:         matchDescript = matchingDescript(entry, Repos)
31:         /*need to check DM for aliases in addDescript*/
32:         eurm.addDescript(matchDescript, Repos)
33:      end for
34:   end if
35:   if entryType.equals(POSTCOND) then
36:      /*similar to precond, see lines 17 to 25*/
37:   end if
38: end for
39: preproc_URM (URM, ref Repos))
   /*Input: Same as in Line 1
   Output: Repos - Repository updated with artifacts from
                   NLP and DM */
40: for each entry ∈ URM do
41:    /*Uses POS tagging and PennTreebank Project*/
```

```
    anpos ← create(entry) /*returns annotated pos*/
    /*Calls APIs in OpenIE pipeline*/
    anobjs ← pipeline(apos) /*returns annotated objects*/
    ansros ← generate(anobjs) /*returns annotated SROs*/
    Repos.add(entry, ansros)
end for
for each field ∈ DM do
    dmgraph ← addToGraph(field) /*creates bidirectional graph*/
end for
Repos.add(dmgraph)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="EnhancedRDS" nsURI="http://org/eclipse/example/Require" nsPrefix="">
  <eClassifiers xsi:type="ecore:EClass" name="UserReq" eSuperTypes="#//EURMM">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="URID" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="description" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="actoruserreq" eType="#//Actor"
        containment="true" eOpposite="#//Actor/eerdsmodel"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="preconuserreq" eType="#//Precondition"
        containment="true" eOpposite="#//Precondition/eerdsmodel"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="postconuserreq" eType="#//Postcondition"
        containment="true" eOpposite="#//Postcondition/eerdsmodel"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="triguserreq" eType="#//Trigger"
        containment="true" eOpposite="#//Trigger/eerdsmodel"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Actor">
    <eStructuralFeatures xsi:type="ecore:EReference" name="table" upperBound="-1"
        eType="#//Table" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="actor" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="eerdsmodel" eType="#//UserReq"
        eOpposite="#//UserReq/actoruserreq"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="alias" eType="#//Alias"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Precondition">
    <eStructuralFeatures xsi:type="ecore:EReference" name="eerdsmodel" eType="#//UserReq"
        eOpposite="#//UserReq/preconuserreq"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="attributes" eType="#//Attributes"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="conprecon" upperBound="-1"
        eType="#//Constraints" containment="true" eOpposite="#//Constraints/precondition"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Constraints">
    <eStructuralFeatures xsi:type="ecore:EReference" name="precondition" eType="#//Precondition"
        eOpposite="#//Precondition/conprecon"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="csubcon" eType="#//CSubject"
        containment="true" eOpposite="#//CSubject/constraintssub"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="crelcon" eType="#//CRelation"
        containment="true" eOpposite="#//CRelation/constraintsrel"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="cobjcon" eType="#//TS"
```

Figure A2: Enhanced User Requirement Meta-Model XMI - Part 1.

```
44          containment="true" eOpposite="#//TS/constraintsobj"/>
45        <eStructuralFeatures xsi:type="ecore:EReference" name="postcondition" eType="#//Postcondition"
46          eOpposite="#//Postcondition/conpostcon"/>
47        <eStructuralFeatures xsi:type="ecore:EReference" name="cpagecon" eType="#//CPage"
48          containment="true" eOpposite="#//CPage/constraintspage"/>
49      </eClassifiers>
50      <eClassifiers xsi:type="ecore:EClass" name="Trigger">
51        <eStructuralFeatures xsi:type="ecore:EReference" name="eerdsmodel" eType="#//UserReq"
52          eOpposite="#//UserReq/triguserreq"/>
53        <eStructuralFeatures xsi:type="ecore:EReference" name="trigstep" upperBound="-1"
54          eType="#//Steps" containment="true" eOpposite="#//Steps/steptrig"/>
55      </eClassifiers>
56      <eClassifiers xsi:type="ecore:EClass" name="Steps">
57        <eStructuralFeatures xsi:type="ecore:EReference" name="steptrig" eType="#//Trigger"
58          eOpposite="#//Trigger/trigstep"/>
59        <eStructuralFeatures xsi:type="ecore:EReference" name="tssubstep" eType="#//TSSubject"
60          containment="true" eOpposite="#//TSSubject/step"/>
61        <eStructuralFeatures xsi:type="ecore:EReference" name="tsactstep" eType="#//TSAction"
62          containment="true" eOpposite="#//TSAction/step"/>
63        <eStructuralFeatures xsi:type="ecore:EReference" name="tsdeststep" eType="#//TSDest"
64          containment="true" eOpposite="#//TSDest/step"/>
65      </eClassifiers>
66      <eClassifiers xsi:type="ecore:EClass" name="Table">
67        <eStructuralFeatures xsi:type="ecore:EReference" name="table" eType="#//Table"/>
68        <eStructuralFeatures xsi:type="ecore:EReference" name="attributes" eType="#//Attributes"/>
69      </eClassifiers>
70      <eClassifiers xsi:type="ecore:EClass" name="Attributes">
71        <eStructuralFeatures xsi:type="ecore:EAttribute" name="columnnames"
72        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
73      </eClassifiers>
74      <eClassifiers xsi:type="ecore:EClass" name="Alias">
75        <eStructuralFeatures xsi:type="ecore:EReference" name="steps" eType="#//Steps"/>
76        <eStructuralFeatures xsi:type="ecore:EAttribute" name="aliastext"
77        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
78      </eClassifiers>
79      <eClassifiers xsi:type="ecore:EClass" name="TSAction">
80        <eStructuralFeatures xsi:type="ecore:EReference" name="step" eType="#//Steps"
81          eOpposite="#//Steps/tsactstep"/>
82        <eStructuralFeatures xsi:type="ecore:EAttribute" name="tsAction"
83        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
84      </eClassifiers>
85      <eClassifiers xsi:type="ecore:EClass" name="Noun">
86        <eStructuralFeatures xsi:type="ecore:EAttribute" name="NN"
```

Figure A3: Enhanced User Requirement Meta-Model XMI - Part 2.

```
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="actor" eType="#//Actor"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="EURMM"/>
  <eClassifiers xsi:type="ecore:EClass" name="UserStory" eSuperTypes="#//EURMM"/>
  <eClassifiers xsi:type="ecore:EClass" name="CSubject">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="cSubject"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="constraintssub" eType="#//Constraints"
        eOpposite="#//Constraints/csubcon"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="CRelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="cRelation"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="constraintsrel" eType="#//Constraints"
        eOpposite="#//Constraints/crelcon"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TS">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="cObject"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="constraintsobj" eType="#//Constraints"
        eOpposite="#//Constraints/cobjcon"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Postcondition">
    <eStructuralFeatures xsi:type="ecore:EReference" name="eerdsmodel" eType="#//UserReq"
        eOpposite="#//UserReq/postconuserreq"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="conpostcon" upperBound="-1"
        eType="#//Constraints" containment="true" eOpposite="#//Constraints/postcondition"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TSSubject">
    <eStructuralFeatures xsi:type="ecore:EReference" name="step" eType="#//Steps"
        eOpposite="#//Steps/tssubstep"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tsSubject"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TSDest">
    <eStructuralFeatures xsi:type="ecore:EReference" name="step" eType="#//Steps"
        eOpposite="#//Steps/tsdeststep"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tsDest"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="CPage">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pageName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="constraintspage" eType="#//Constraints"
        eOpposite="#//Constraints/cpagecon"/>
  </eClassifiers>
</ecore:EPackage>
```

Figure A4: Enhanced User Requirement Meta-Model XMI - Part 3.

```
<?xml version="1.0" encoding="UTF-8"?><UserReq xmlns="http://org/eclipse/example/Require"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
URID="PMS_02_ login-Employee"
description="Employee enters his/her username and password to login to the system"
xmi:version="2.0" xsi:schemaLocation="http://org/eclipse/example/Require
../../src/main/java/EnhancedMetamodel/EnhancedRDS.ecore">
  <actoruserreq actor="Employee"/>
  <preconuserreq>
    <conprecon>
      <csubcon cSubject="Employee"/>
      <crelcon cRelation="has"/>
      <cobjcon cObject="User.User_ID"/>
      <cpagecon pageName="null"/>
    </conprecon>
    <conprecon>
      <csubcon cSubject="Employee"/>
      <crelcon cRelation="has"/>
      <cobjcon cObject="User.User_PWD"/>
      <cpagecon pageName="null"/>
    </conprecon>
    <conprecon>
      <csubcon cSubject="Employee"/>
      <crelcon cRelation="is"/>
      <cobjcon cObject="null"/>
      <cpagecon pageName="login page"/>
    </conprecon>
  </preconuserreq>
  <postconuserreq>
    <conpostcon>
      <csubcon cSubject="pms system"/>
      <crelcon cRelation="redirected"/>
      <cobjcon cObject="employee homepage"/>
      <cpagecon/>
    </conpostcon>
  </postconuserreq>
  <triguserreq>
    <trigstep>
      <tssubstep tsSubject="Employee"/>
      <tsactstep tsAction="enters"/>
      <tsdeststep tsDest="User.User_ID"/>
    </trigstep>
    <trigstep>
      <tssubstep tsSubject="Employee"/>
      <tsactstep tsAction="enters"/>
      <tsdeststep tsDest="User.User_PWD"/>
    </trigstep>
    <trigstep>
      <tssubstep tsSubject="Employee"/>
      <tsactstep tsAction="clicks"/>
      <tsdeststep tsDest="signin button"/>
    </trigstep>
  </triguserreq>
</UserReq>
```

Figure A5: Enhanced User Requirements Model for Use Case Login

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="Abstracttc"
    nsURI="http://org/eclipse/example/Require">
  <eClassifiers xsi:type="ecore:EClass" name="ATC" eSuperTypes="#//ATCMM">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="testcaseID"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="expectedoutputtestcase"
        upperBound="-1" eType="#//ExpectedOutput" containment="true"
        eOpposite="#//ExpectedOutput/testcaseexpectedoutput"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="testsetuptestcase" upperBound="-1"
        eType="#//TestSetup" containment="true" eOpposite="#//TestSetup/testcasetestsetup"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="purposetestcase" eType="#//Purpose"
        containment="true" eOpposite="#//Purpose/testcase"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ieopairs" upperBound="-1"
        eType="#//IEOpairs" containment="true" eOpposite="#//IEOpairs/atc"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ExpectedOutput">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="expectedoutput" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="testcaseexpectedoutput"
        eType="#//ATC" eOpposite="#//ATC/expectedoutputtestcase"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tconstraint" upperBound="-1"
        eType="#//TConstraint" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TestSetup">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="testsetup" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="testcasetestsetup" eType="#//ATC"
        eOpposite="#//ATC/testsetuptestcase"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tconstraint" upperBound="-1"
        eType="#//TConstraint" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Purpose">
    <eStructuralFeatures xsi:type="ecore:EReference" name="testcase" eType="#//ATC"
        eOpposite="#//ATC/purposetestcase"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="purpose" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ATCMM"/>
  <eClassifiers xsi:type="ecore:EClass" name="DataModel" eSuperTypes="#//ATCMM">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dmName"
```

Figure A6: Abstract Test Case Meta-Model XMI - Part 1.

```
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_table" upperBound="-1"
        eType="#//DM_Table" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TConstraint">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="setupID"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="subject" eType="#//Subject"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DM_Constr" eSuperTypes="#//TConstraint">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dmconstr"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="em_rel_constr" upperBound="-1"
        eType="#//DM_REL_Constr" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_table" upperBound="-1"
        eType="#//DM_Table" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="page_fields" eType="#//Page_Fields"
        containment="true" eOpposite="#//Page_Fields/dm_constr"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Sys_Constr" eSuperTypes="#//TConstraint">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="sconstr"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="SysPage" eSuperTypes="#//Sys_Constr">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="syspage"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="page_fields" upperBound="-1"
        eType-"#//Page_Fields" containment-"true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="state"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ATC_Flow">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="flow"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="atc_actor" upperBound="-1"
        eType="#//ATC_Actor" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="atc_action" upperBound="-1"
        eType="#//ATC_Action" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="atc_entity" upperBound="-1"
        eType="#//ATC_Entity" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ATC_Entity">
```

Figure A7: Abstract Test Case Meta-Model XMI - Part 2.

```
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="entityName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_table" upperBound="-1"
        eType="#//DM_Table" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ATC_Action">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="action"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ATC_Actor">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="actorName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Subject">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="subject"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="IEOpairs">
    <eStructuralFeatures xsi:type="ecore:EReference" name="atc"
    eType="#//ATC" eOpposite="#//ATC/ieopairs"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="atc_flow" upperBound="-1"
        eType="#//ATC_Flow" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="expectedoutput" upperBound="-1"
        eType="#//ExpectedOutput" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DM_REL_Constr">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="rel" eType="#//DM_Rel"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="relation"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DM_Table">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tabName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_attribute" upperBound="-1"
        eType="#//DM_Attribute" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DM_Attribute">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="colName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Page_Fields">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="fieldName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

Figure A8: Abstract Test Case Meta-Model XMI - Part 3.

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_table" upperBound="-1"
        eType="#//DM_Table" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm_constr" eType="#//DM_Constr"
        eOpposite="#//DM_Constr/page_fields"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="DM_Rel">
    <eLiterals name="Instance_of"/>
    <eLiterals name="Has" value="1"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Sys_Session" eSuperTypes="#//Sys_Constr">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="session"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

Figure A9: Abstract Test Case Meta-Model XMI - Part 4.

```
1  -- @path EUR=/RDT/src/main/java/EnhancedMetamodel/EnhancedRDS.ecore
2  -- @path ATC=/RDT/src/main/java/AbstractTCMetamodel/Abstracttc.ecore
3
4  module E2A;
5  create OUT : ATC from IN : EUR;
6  helper context EUR!Precondition def: allclasses(): Sequence(EUR!Precondition)=
7      self.conprecon->iterate(e; acc:Sequence(EUR!Precondition)=Sequence{}|
8      acc->union(Set{e}));
9
10
11 helper context EUR!Postcondition def: allclasses3(): Sequence(EUR!Postcondition)=
12     self.conpostcon->iterate(e; acc:Sequence(EUR!Postcondition)=Sequence{}|
13     acc->union(Set{e}));
14
15
16 helper context EUR!Trigger def: allclasses2(): Sequence(EUR!Trigger)=
17     self.trigstep->iterate(e; acc:Sequence(EUR!Trigger)=Sequence{}|
18     acc->union(Set{e}));
19
20 helper context EUR!CPage def: isPagedefined(): Boolean =
21     if not self.constraintspage.oclIsUndefined() then
22         true
23         else
24             false
25     endif;
26
27 rule Enhanced2Abstractiddesc{
28     from
29         e:EUR!UserReq(true)
30     to
31         x:ATC!ATC(testcaseID<-e.URID),
32         y:ATC!Purpose(purpose<-e.description)}
33
34 lazy rule CSubject2Subject{
35     from
36         e:EUR!Constraints(true)
37     to
38         x:ATC!Subject(subject<-e.csubcon.cSubject)}
```

Figure A10: ATL Rules - Part 1.

```
40
41 lazy rule CRelation2DM_REL_Constr{
42     from
43         e:EUR!Constraints(true)
44     to
45         y:ATC!DM_REL_Constr(relation<-e.crelcon.cRelation)}
46
47 lazy rule TS2DM_Table{
48     from
49         e:EUR!TS(true)
50     to
51         y:ATC!DM_Table(tabName<-e.cobjcon.cObject)}
52
53 lazy rule CPage2Page_Fields{
54     from
55
56         e:EUR!Constraints(true)
57     to
58     x:ATC!Page_Fields(fieldName<-e.cpagecon.pageName)}
59
60 lazy rule Constraints2TConstraint{
61     from
62         e:EUR!Constraints(true)
63     to
64         x:ATC!DM_Constr(subject<-thisModule.CSubject2Subject(e),
65             em_rel_constr<-thisModule.CRelation2DM_REL_Constr(e),
66             dm_table<-thisModule.TS2DM_Table(e),
67             page_fields<-thisModule.CPage2Page_Fields(e))}
68
69 lazy rule Constraints2TConstrainta{
70     from
71         e:EUR!Constraints(true)
72     to
73         x:ATC!Sys_Constr(
74             page_fields<-thisModule.CPage2Page_Fields(e))}
75
```

Figure A11: ATL Rules - Part 2.

```
lazy rule TSSubject2ATC_Actor{
    from
        e:EUR!TSSubject(true)
    to
        x:ATC!ATC_Actor(actorName<-e.tssubstep.tsSubject)}

lazy rule TSAction2ATC_Action{
    from
        e:EUR!TSAction(true)
    to
        x:ATC!ATC_Action(action<-e.tsactstep.tsAction)}

lazy rule TSDest2ATC_Entity{
    from
        e:EUR!TSDest(true)
    to
        x:ATC!ATC_Entity(entityName<-e.tsdeststep.tsDest)}

lazy rule Steps2ATC_Flow{
    from
        e:EUR!Steps(true)
    to
        x:ATC!ATC_Flow(atc_actor<-thisModule.TSSubject2ATC_Actor(e),
            atc_action<-thisModule.TSAction2ATC_Action(e),
            atc_entity<-thisModule.TSDest2ATC_Entity(e))}
```

Figure A12: ATL Rules - Part 3.

```
rule Precondition2Testsetup{
    from
        e:EUR!Precondition(true)
    to
        x:ATC!TestSetup(tconstraint<- e.allclasses()->
            collect(a | thisModule.Constraints2TConstraint(a))) }


rule Trigger2IEOPairs{
    from
        e:EUR!Trigger(true)
    to
        x:ATC!IEOpairs(atc_flow<-e.allclasses2()->
            collect(a|thisModule.Steps2ATC_Flow(a)))}


rule Postcondition2ExpectedOutput{
    from
        e:EUR!Postcondition(true)
    to
        x:ATC!ExpectedOutput(tconstraint<- e.allclasses3()->
            collect(a | thisModule.Constraints2TConstraint(a)))}
```

Figure A13: ATL Rules - Part 4.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://org/eclipse/example/Require">
  <ATC testcaseID="PMS_02_ login-Employee"/>
  <Purpose purpose="Employee enters his/her
   username and password to login to the system"/>
  <TestSetup>
    <tconstraint xsi:type="DM_Constr">
      <subject subject="Employee"/>
      <em_rel_constr relation="has"/>
      <dm_table tabName="User.User_ID"/>
      <page_fields fieldName="null"/>
    </tconstraint>
    <tconstraint xsi:type="DM_Constr">
      <subject subject="Employee"/>
      <em_rel_constr relation="has"/>
      <dm_table tabName="User.User_PWD"/>
      <page_fields fieldName="null"/>
    </tconstraint>
    <tconstraint xsi:type="DM_Constr">
      <subject subject="Employee"/>
      <em_rel_constr relation="is"/>
      <dm_table tabName="null"/>
      <page_fields fieldName="login page"/>
    </tconstraint>
  </TestSetup>
  <ExpectedOutput>
    <tconstraint xsi:type="DM_Constr">
      <subject subject="pms system"/>
      <em_rel_constr relation="redirected"/>
      <dm_table tabName="employee homepage"/>
      <page_fields/>
    </tconstraint>
  </ExpectedOutput>
  <IEOpairs>
    <atc_flow>
      <atc_actor actorName="Employee"/>
      <atc_action action="enters"/>
      <atc_entity entityName="User.User_ID"/>
    </atc_flow>
    <atc_flow>
      <atc_actor actorName="Employee"/>
      <atc_action action="enters"/>
      <atc_entity entityName="User.User_PWD"/>
    </atc_flow>
    <atc_flow>
      <atc_actor actorName="Employee"/>
      <atc_action action="clicks"/>
      <atc_entity entityName="signin button"/>
    </atc_flow>
  </IEOpairs>
</xmi:XMI>
```

Figure A14: Abstract Test Case Model for Use Case Login

VITA

CHAITHRA SAI ALLALA

Academics

2014	Bachelor of Science in Electronics and Computer Science Sreenidhi Institute of science and technology India
2015	Master of Science in Computer and information Systems School of Computer Information Systems Florida International University Miami, Florida
2023	Doctor of Philosophy in Computer and information Systems School of Computer Information Systems Florida International University Specialization Software Engineering

Publications

1. Ellen L Brown, Nicole Ruggiano, Sai Chaithra Allala, Peter J. Clarke, Debra Davis, Lisa Roberts, C. Victoria Framil, Mariateresa H Munoz, Monica Strauss Hough, Michelle Suzanne Bourgeois. A Memory and Communication App for Persons Living with Dementia: It takes a Village! submitted to Journal of Medical Internet Research (Nov 2, 2022). http://doi.org/10.2196/preprints.44007
2. Juan P. Sotomayor, Sai Chaithra Allala, Dionny Santiago, Tariq M. King, and Peter J. Clarke, Comparison of open-source runtime testing tools for microservices, Software Quality Journal, pp. 1–33, 2022.
3. Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago Tariq M. King and Peter J. Clarke. Generating Abstract Test Cases from User Requirements using MDSE and NLP. 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS) , 2022, p.1-753
4. Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago, Tariq M. King, and Peter J. Clarke. Towards transforming user requirements to test cases using mde and nlp. In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), volume 2, pages 350-355, 2019.

5. Juan P. Sotomayor, Sai Chaithra Allala, Patrick Alt, Justin Phillips, Tariq M. King, and Peter J. Clarke. Comparison of runtime testing tools for microservices. In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), volume 2, pages 356-361, 2019.
6. Nicole Ruggiano, Ellen Brown, C Victoria Framil, Shannon Hurley, Lisa Roberts, Peter Clarke, Sai Chaithra Allala, Jane Daquin. Supporting Dementia Caregivers During COVID-19 with CareHeroes IT: Usage Patterns and Outcomes. In November 2022 Gerontological Society America Journals.
7. Ellen L Brown, Nicole Ruggiano, Lisa Roberts, Peter J Clarke, Debra J Davis, Monica Strauss Hough, Mariateresa (Teri) H Muñoz, C Victoria Framil, Sai Chaithra Allala. Integration of Health Information Technology and Promotion of Personhood in Family- Centered Dementia Care. Conference: AAIC 2021 Alzheimer's Association International Conference (July 2021).
8. Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago and Peter J. Clarke. An Approach for Automatically Generating Test Cases from User Requirements Using MDSE and NLP. In 2019 Pacific Northwest Software Quality Conference (PNSQC).
9. Juan P. Sotomayor, Sai Chaithra Allala, Patrick Alt, Justin Phillips. A Survey of Open Source Tools Used for Testing Microservices. In 2019 Pacific Northwest Software Quality Conference (PNSQC).
10. Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago, Tariq M. King, and Peter J. Clarke. From User Requirements to Abstract Test Cases using MDE and NLP.Journal to be submitted.