Generating Abstract Test Cases from User Requirements using MDSE and NLP

Sai Chaithra Allala[1], Juan P. Sotomayor[1], Dionny Santiago[1], Tariq M. King[2], and Peter J. Clarke[1]
[1]School of Computing and Information Sciences , Florida International University, Miami, FL 33199, USA
[2]EPAM, 41 University Drive Suite 202, Newton PA, 18940, USA
{salla010, jsoto128, dsant005}@cis.fiu.edu, tariq_king@epam.com, clarkep@cis.fiu.edu

***Abstract*—Model-driven software engineering (MDSE) has emerged as a popular and commonly used method for designing software systems in which models are the primary development artifact over the last decade. MDSE has resulted in the trend toward further automating the software process. However, the generation of test cases from user requirements still lags in reaching the required level of automation. Given that most user requirements are written in natural language, the recent advances in natural language processing (NLP) provide an opportunity to further automate the test generation process.**

In this paper, we exploit the advances in MDSE and NLP to generate abstract test cases from user requirements written in structured natural language and the respective data model. We accomplish this by creating meta-models for user requirements and abstract test cases and defining t he appropriate transformation rules. To support this transformation, helper methods are defined t o e xtract t he r elevant i nformation from user requirements related to testing. To show the feasibility of the approach, we developed a prototype and conducted a case study with use cases and test cases from a Payroll Management System.

Keywords*—Abstract Test Cases; Model-Driven Software Development; Natural Language Processing; Software Testing; User Requirements.

I. Introduction

Testing continues to be the major approach to ensuring software quality during development. 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 [1]. An essential aspect of the software process is the consistency of the artifacts developed starting from the requirements through software design and implementation [2]. User requirements, written as use cases or user stories, are the basis for developing system test cases and help the developer to answer the following question. "Are we building the right product?" Software testing is becoming increasingly challenging as the complexity of systems continues to evolve and the deployment of applications becomes more demanding [3], [4].

During the past decade, there have been many attempts to automate the generation of test cases from user requirements (formal or informal) [5]–[9] with some success. 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 I ntelligence (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.

In this paper, we extend the work presented by Allala et al. [10] that introduces an approach to automatically generating test cases from user requirements using MDSE and NLP. These extensions include the following. (1) Providing more details on the overall process that generates test cases from use cases or user stories. (2) Describing meta-models for enhanced user requirements (EURs)(user requirements and data models) and abstract test cases (ATCs). (3) Describing the underlying algorithm used in the model-to-model (M2M) transformation from the EUR meta-model to the ATC meta-model.

The underlying concepts of model transformations are a vital ingredient to our approach. Using our automated approach, we attempt to answer two questions: (1) How similar are the ATCs generated using our automated approach compared to the ATCs generated using a manual approach? (2) What are the major limitations of using our approach to generate ATCs? The contributions of the work presented in the paper include the following:

- Meta-models for enhanced user requirements (EURs) and abstract test cases (ATCs).
- An algorithm that generates a EUR model from user requirements (use cases) and a data model (ER model).
- A transformation that generates ATCs from EUR models.
- Results of a case study that compares manually generated ATCs against the ATCs generated from the prototype using our approach.

The paper is organized as follows. Section II provides background and related work. Section III describes our high-level approach to transforming user requirements to test cases and introduces an illustrative example. Section IV describes the EUR meta-model and generation EUR models. Section V describes the transformation from EUR models to ATCs. We then present a case study in Section VI and conclude the paper in Section VII.

II. Literature Review

In this section, we provide a brief introduction to Model-Driven Software Engineering (MDE) and Natural Language Processing (NLP) and compare our work to the most closely related work in the literature.

A. Background

A *meta-model* can be defined as an abstraction of a model, highlighting the properties of the model itself [11]. If a model

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00080

adheres to the properties of the meta-model, then the model conforms to the meta-model.

A *transformation* includes a set of rules that allows to create a target model (M_2) from a source model (M_1). The rules defined by the transformation are at the level of the meta-models.

Natural Language Processing (NLP), a component of Artificial Intelligence and Computational Linguistics [5], has the goal of performing human-like language processing. NLP requires several levels of language processing that may interact with each other [12].

One of the most widely used tools for NLP is Stanford CoreNLP [13], which is a toolkit that provides an extensible pipeline that provides core natural language analysis. The Stanford CoreNLP consists of several components including 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; 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.

B. Related Work

The research literature contains a broad spectrum of work on automatically generating test cases using MDSE [14], and more recently using NLP [5]. 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.

MDSE: Gutiérrez et al. [14] present an approach to generating functional test cases from functional 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. [15] propose 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.

NLP: Garousi et al. [5] 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.

As previously stated, the most common tool used found in the SLM was the Stanford CoreNLP [13]. 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. [16] 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.

Similar to the approaches by Garousi et al. [5] and Thummalapenta et al. [16], 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.

MDE and NLP:

Wang et al. [9] 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

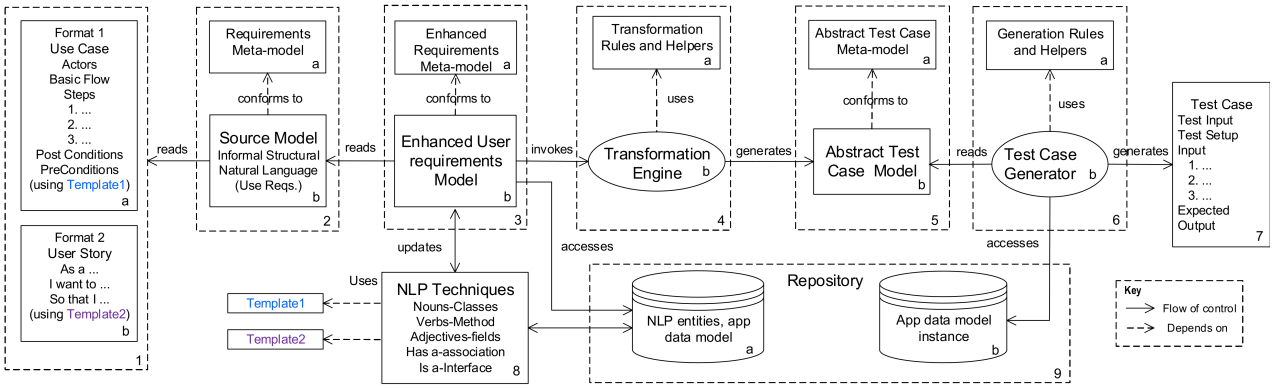

Figure 1. Overview of the high-level transformation process.

testing. UMTG uses NLP to build Use Case Test Models (UCTMs) from the RUCM specifications. UMTG uses five NLP analysis to extract the pre, post and guard conditions (constraints) needed to generate the test inputs for the test cases. 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.

Our approach is closer to that of Wang et al. [9] 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. uses 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. 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.

III. Overview of Approach

In this section, we provide an overview of our approach to transforming user requirements to test cases while discussing an example use case and the related entity-relationship model.

A. High-level Transformation

Figure 1 shows a high-level view of the transformation process from user requirements to test case generation. Figure 1 is updated from the previous published work [10] and uses additional models and processes in the transformation. The following enumerated list describes each of the major phases in the overall transformation process.

(1) The source for the transformation process is either a use case (upper left side of Figure 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 [10]. *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) [17].

(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 [18].

(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

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

Figure 2. Example of the login use case for the *Payroll Management System (PMS)*.

toolkit [5] 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.

In this paper, we describe the models and process from phase 2, the reading of the user requirements, to phase 5, the generation of the abstract test cases, as shown in Figure 1. Only the use case format for user requirements is used.

B. Example of Use Case 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 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.

We use a trivial use case as the example, that is, logging into the PMS system, as shown in Figure 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. 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 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.

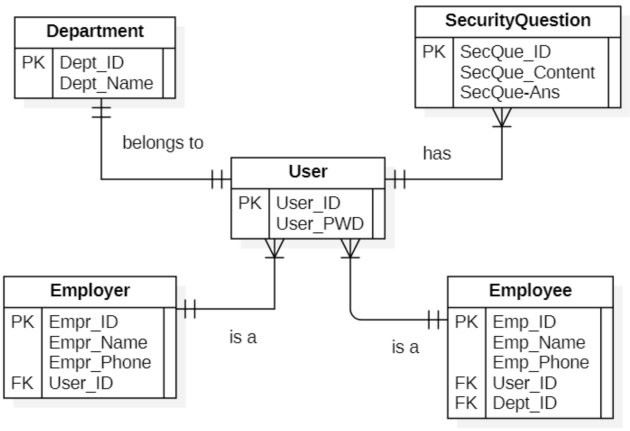

Figure 3. Partial entity-relation diagram (ERD) for PMS.

IV. Enhanced User Requirements Model

This section details how the Enhanced User Requirements Model (EURM) is generated from the user requirements model, data model, and the entities generated during NLP.

A. Enhanced User Requirements Meta-model

To support the transformation process shown in Figure 1 we define a meta-model for the EURM. This meta-model is shown in the box labeled 3(a) in Figure 1. A partial representation of the enhanced user requirements meta-model, referred to as `EUR-MM`, is shown in Figure 4. `EUR-MM` consists of `UserReq` - the user requirements meta-model derived from the meta-model described in Allala et al. [10] and a `DataModel` that represents parts of the meta-model for an enhanced entity-relation model [19].

The `UserReq`, showing in Figure 4 shows 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.

Not shown in Figure 4 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`

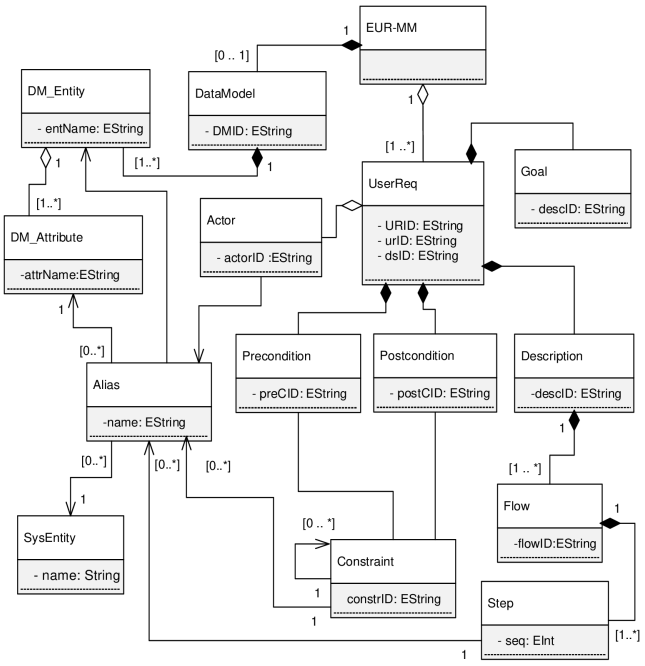

Figure 4. Partial enhanced user requirements (EUR) meta-model.

- 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 is shown on the left side of Figure 6.

B. 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 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 [5] 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. This mapping approach supports the substring matching for all the matching procedures.

Algorithm 1 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 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*/
42:    anpos ← create(entry) /*returns annotated pos*/
43:    /*Calls APIs in OpenIE pipeline*/
44:    anobjs ← pipeline(apos) /*returns annotated objects*/
45:    ansros ← generate(anobjs) /*returns annotated SROs*/
46:    Repos.add(entry, ansros)
47: end for
48: for each field ∈ DM do
49:    dmgraph ← addToGraph(field) /*creates bidirectional graph*/
50: end for
51: Repos.add(dmgraph)
```

The preprocessing 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 objects (SROs), `ansros`, and added to the repository along with the entry. The preprocessing 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 as output a EURM and the updated repository. 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 preprocessing procedure is called. Lines 5 through 10 check 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 IV-A 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 in Figure 2 is shown on the left side of Figure 6.

V. Transforming EURs to Abstract Test Cases

In this section we describe how EURMs are transformed into abstract test cases (ATCs). This description includes the ATC meta-model and the Atlas Transformation Language (ATL) rules in Figure 5 used during the transformation process. The transformation process is shown in the boxes labeled 3, 4 and 5 in Figure 1.

A. Abstract Test Case 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. The ATC 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 (`Constraint`). 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.

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 also enumerated types. For each enumerated type we show the initial set i.e `Input_Action` (enters,click_on,select) which may be expanded upon in future work. An instance of the ATC meta-model is shown on the right side of Figure 6. Due to page length restriction the ATC meta-model diagram was not included in this paper.

B. Transformation rules

An ATL 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, and the target meta-model (ATC) elements mentioned above. Each rule specifies how the source model elements must be matched and navigated in order to initialize target model elements. The transformation processes uses declarative rules for `testcaseID`, `purpose` and so on.

Figure 5 shows several rules used in the transformation engine. In this example, the execution starts on line 16 with the rule `Precondition2Testsetup`, which iterates through each precondition entry and generates an entry for test setup by calling the lazy rule `Constraints2TConstraint`, line 5. 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

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

Figure 5. Atlas transformation language.

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 18 in Figure 5. ATL can manipulate collections similar to OCL. The right side of the left arrow may also use helper functions to assist with processing collections.

Helper functions can be used to define (global) variables and functions. They can call each other (recursion is possible) or they can be called from within rules. In the example in Figure 5 we define a helper function, lines 1 to 3, which iterates over each precondition entry and this allows the target pattern `ATC!TestSetup` to invoke the rule `Constraints2TConstraint` for each precondition entry.

C. Illustrative Example of Transformation

In Figure 6 we show an example of transformation from EURM to ATC. On the left side of the figure we show the EURM that is created from the use case shown in Figure 2. The right side shows the ATC generated using the transformation process described in this section, Boxes 3, 4, and 5 in Figure 1. Starting at the top of Figure 6, 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 case is Employee and is represented as `User` in our data model we use the above algorithm in Section IV-B 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`, `CEntity` and `SDest` (optional). The first entry in the precondition (*Employee has User.User_ID and User.User_PWD*) is transformed into the first two entries of the test setup (*1. Employee has User.User_ID* and *2. Employee has User.User_PWD*).

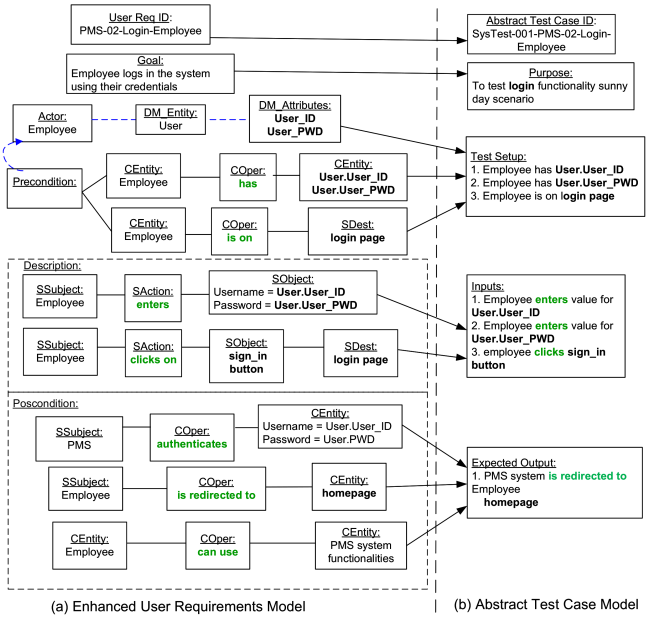

Figure 6. Illustration showing the transformation of an enhanced user requirements model to an abstract test case model.

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. Employee enters Username = User.User_ID and Password = User.User_PWD* is transformed into the `Inputs` entries (*1. User enters value for User.User_ID*) and (*2. Employee enters value for User.User_PWD*). A similar approach to transforming the precondition is used for the postcondition. The main difference is that the certain keywords are translated into system state.

VI. Case Study

The case study is 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.

A. Setup

As described in Section III-B, the user requirements for the experiments came from a project from a graduate software engineering class at Florida International University (FIU) in Fall 2015. The name of the application 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

TABLE I
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-ApproveTimeSheet	The employer approves the Employee's timesheet submitted information
6	PMS-04-ModifyTimeSheet	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-SearchEmployee	The Employer searches for the Employee's information
9	PMS-08-SaveTimeSheet	Employer saves Employee's timesheet in the system
10	PMS-06-CalculatePay	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

of 28 use cases, of which 12 use cases were implemented for the PMS application. Table I shows the user requirements consisting of 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-02-Login-Employee`), was the illustrative example in the paper.

The abstract test cases (ATCs) used in the case study were generated using the approach shown in Figure 1 and manually by the authors of the paper, 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 VI-B.

B. Processing ATCs

Manually Generating ATCs: The research team, excluding the person implementing the automated approach in this paper, 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 used. Before creating the ATCs, the team members reviewed the complete meta-model for ATCs; a partial meta-model mentioned in Section V-A. 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 I, the two team members compared their ATCs for any discrepancies. Discrepancies were resolved by a third team member, independent of the lead author, to create a single ATC.

Automatically Generating ATCs: A prototype was developed using the Eclipse Modeling Framework (EMF) [20] to generate the ATCs from user requirements, similar to the use case shown in Figure 2. The prototype 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. EMF [20] is a modeling framework and code generation framework for building tools and other applications. EMF uses the Ecore modeling language. OCLinEcore [21] provides a textual concrete syntax that makes both Ecore and OCL accessible to users. As previously stated, the Stanford-NLP [5] toolkit was used for the natural language analysis, see Section IV-B. The intermediate models generated in the prototype used the XML Metadata Interchange (XMI), a standard for exchanging metadata information [22].

Comparing ATCs: The comparison of the manually and automatically generated ATCs were done based on the structure of the meta-model for ATCs. 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 data model, state of the data model, actors, input actions, system state (sessions), pages (types, fields, and buttons), and output actions.

C. 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 II shows the results obtained when comparing the manually and automatically generated ATCs. The columns from left to right in Table II starting at Column 3 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.

We use the following formulas when computing the *Precision*, *Recall* and *F1 Score* [23].

$$Precision\ (Pre) = TP/(TP + FP) \tag{1}$$

$$Recall\ (Rec) = TP/(TP + FN) \tag{2}$$

TABLE II
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

$$F1\ Score = 2 * (Pre * Rec)/(Prec + Rec) \quad (3)$$

Row 1 of Table II shows the data collected when the manually generated ATC for the use case *PMS-01-Login-Employee* is compared to the automatically generated ATC from the prototype. In the test setup there are 3 subject (*Subj*) entries (`Employee`), 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 6. 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., `Employee enters value of User.User_ID in the user_id field`. Using equations 1 to 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 II 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 employee information. Row 8, showing the search employee 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.

D. 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 II. 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 II 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 II. For example, the data from first pass for the login use case in Row 1 have false positives - 8, false negatives - 5, Precision - 0.62, Recall - 0.72 and the F1 score - 0.67. To obtain the values in Table II 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, automat-

ically 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.

VII. Conclusion and Future Work

In this paper, we present our approach that automatically transforms user requirements into abstract test cases (ATCs) using model-driven software engineering (MDSE) and natural language processing (NLP). The approach begins with the construction of an enhanced user requirements (EUR) model. The EUR 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 ATCs, we developed meta-models for EUR models and ATCs. A prototype was developed to perform the transformation 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 4 undergraduate software projects. The ATCs generated using the prototype 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. In the future, we plan to extend this work to generate concrete test cases from the ATCs using an instance of the data model (database) and one or more test generation techniques, e.g., boundary value analysis.

References

[1] D. Kumar and K. Mishra, "The impacts of test automation on software's cost, quality and time to market," *Procedia Computer Science*, vol. 79, pp. 8–15, 12 2016.

[2] T. M. Hangensen and B. B. Kristensen, "Consistency in software system development: Framework, model, techniques, & tools," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 5, pp. 58–67, Nov. 1992. [Online]. Available: http://doi.acm.org/10.1145/142882.142914

[3] J. P. Sotomayor, S. C. Allala, D. Santiago, T. M. King, and P. J. Clarke, "Comparison of open-source runtime testing tools for microservices," *Software Quality Journal*, pp. 1–33, 2022.

[4] D. Smith, D. Villalba, M. Irvine, D. Stanke, and N. Harvey, "Accelerate state of devops report," DORA & Google Cloud, 2021, https://services.google.com/fh/files/misc/state-of-devops-2021.pdf.

[5] V. Garousi, S. Bauer, and M. Felderer, "NLP-assisted software testing: A systematic mapping of the literature," *Information and Software Technology*, vol. 126, pp. 1–20, 2020, paper no. 106321. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300744

[6] K. Jin and K. 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)*, ser. ISEC 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3452383.3452408

[7] M. Kassab, J. F. DeFranco, and P. A. Laplante, "Software testing: The state of the practice," *IEEE Software*, no. 5, pp. 46–52, 2017.

[8] D. Santiago, J. Phillips, P. Alt, B. Muras, T. M. King, and P. J. Clarke, "Machine learning and constraint solving for automated form testing," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 217–227.

[9] 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*, vol. 48, no. 02, pp. 585–616, feb 2022.

[10] S. C. Allala, J. P. Sotomayor, D. Santiago, T. M. King, and P. J. Clarke, "Towards transforming user requirements to test cases using mde and nlp," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2019, pp. 350–355.

[11] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 1st ed. Morgan & Claypool Publishers, 2012.

[12] E. D. Liddy, "Natural language processing." in *Encyclopedia of Library and Information Science*. New York, NY: Marcel Decker, Inc., 2001.

[13] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, P. Inc, S. J. Bethard, and D. Mcclosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[14] J. Gutiérrez, M. Escalona, and M. Mejías, "A model-driven approach for functional test case generation," *Journal of Systems and Software*, vol. 109, pp. 214–228, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001703

[15] C. Hue, D.-H. Dang, N. Binh, and H. Truong, "Usltg: Test case automatic generation by transforming use cases," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, pp. 1313–1345, 09 2019.

[16] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 881–891.

[17] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*, J.-M. Bruel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 128–138.

[18] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2017.

[19] R. D. N. Fidalgo, E. M. a. De Souza, S. España, J. B. De Castro, and O. Pastor, "Eermm: A metamodel for the enhanced entity-relationship model," in *Proceedings of the 31st International Conference on Conceptual Modeling*, ser. ER'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 515–524.

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

[21] R. Gerbig, J. Cadavid, and A. S., "The OCLinEcore Language," https://wiki.eclipse.org/OCL/OCLinEcore, 2019, [Online; accessed 10-Jan-2019].

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

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