Automated Generation of Test Cases from Logical Specification of Software Requirements

Richa Sharma[1] and K. K. Biswas[2]
[1]School of Information Technology, IIT Delhi, New Delhi, India
[2]Department of Computer Science and Engineering, IIT Delhi, New Delhi, India

Keywords: Test Cases, Logical Specification, Courteous Logic, Natural Language Processing.

Abstract: The quality of the delivered software relies on rigorous testing performed. However, designing good test cases is a challenging task. The challenges are multi-fold and test-cases design is often delayed towards the end of implementation phase. In this paper, we propose an approach to automatically generate test cases from the logical form of requirements specifications during early phases of software development. Our approach is based on courteous logic representation of requirements. The Knowledge stored in the courteous logic predicates is used to automatically generate the test cases. We evaluate the effectiveness of our generated test-cases through case-studies.

1 INTRODUCTION

Software testing is an important and integral activity of the software development. The testing process entails designing effective test-cases; generating test-data; executing test-cases for the test data and comparing the results of execution against actual results mentioned in test cases (Ammann and Offutt, 2008). Amongst these activities, designing effective test-cases, that can uncover crucial functional faults, remains a key-challenge. Test cases can be derived from requirements specifications, design artefacts or the source code (Ammann and Offutt, 2008). Requirements Specifications provide useful pointers for conducting functional testing; the design artefacts influence architectural testing and, the source code provides technical know-how for test case design as well as for the requisite test data formats. The research effort towards automation of testing has resulted in some very useful tools like JUnit, Visual test, SQA test, Testmate (Incomplete List of Testing tools, n.d.) etc. However, designing functional test cases based on requirements is still a hard problem. Several authors have proposed approaches for designing functional test cases from UML diagrams (Boghdady et al., 2011), (Kansomkeat, Thiket and Offutt, 2010), (Li et al., 2013); from use-case specifications (Heumann, 2001), (Ahlowalia, 2002) and also, from user-stories used in Agile development (Kamalkar et al., 2013). These suggested approaches assist test engineers by providing them with automatically generated test-cases. These test cases can be further refined by manual intervention, if required, thereby reducing the effort and time spent on writing out the test-cases. However, the challenge involved with these approaches arises from the fact that use-cases, user-stories are expressed in Natural Language (NL) and, the UML diagrams also depend on requirements specifications expressed using NL. The inherent ambiguities and inconsistencies, present in NL specifications of requirements, often lead to misinterpretations and difference of understanding between the client and the development team.

In this paper, we propose an approach to generate test-cases automatically from logical specification of requirements to circumvent these challenges. Logical specifications are formal in nature and, have the advantage of automated analysis and validation (Tsai and Weigert, 1991). The adequacy of courteous logic based representations of the software requirements for inconsistency resolution has been shown in (Sharma and Biswas, 2012). We use these representations of requirements for automated test-case generation. Since formal representations are not the preferred form of representation in industry, therefore, we have also proposed semi-automated approach towards the generation of courteous logic form of

Sharma R. and K. Biswas K..
Automated Generation of Test Cases from Logical Specification of Software Requirements.
DOI: 10.5220/0004972902410248
In *Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2014), pages 241-248
ISBN: 978-989-758-030-7

representation of requirements from their corresponding NL representations in our work (Sharma and Biswas, 2012). This increases applicability of courteous logic based requirements representations in industry.

Our approach involving test generation from courteous logic representation of requirements borrows heavily from semantic head-driven approach for NL Generation (Shieber et al., 1989).

As our main focus is not NL generation, we have adopted and modified their approach to formulate functional test cases.

The paper is organized as follows: Section 2 presents an overview of the courteous logic form of requirements specifications along with the related work done. Section 3 presents our approach of automated test-case generation followed by the case study presented in section 4. In section 5, we present discussion and conclusion.

2 RELATED WORK

2.1 Courteous Logic Representation of Requirements

Courteous Logic as proposed by Grosof is a non-monotonic logic form (Grosof, 1997). Courteous Logical representation is an expressive subclass of ordinary logical representation with which we are familiar and, it has got procedural attachments for prioritized conflict handling. First Order Logic (FOL) has not become widely popular for two main reasons: it is pure belief language and, secondly it is logically monotonic; it can not specify prioritized conflict handling which are logically non-monotonic.

Our motivation behind using Courteous Logic representation of requirements is that real-world knowledge and system requirements for any system in real-world correspond to common-sense form of reasoning. This common-sense reasoning is non-monotonic in nature and, therefore there is a need for non-monotonic logic for representing real-world requirements. Of various available forms of non-monotonic logic like default logic (Reiter, 1980), defeasible logic (Nute, 2001), we preferred courteous logic for its simplicity, ease of use and English-like constructs. Our Courteous Logic representations for requirements are based on IBM's CommonRules, available under free trial license from IBM alpha works (Grosof, 2004). In these representations, prioritization is handled by assigning optional labels to the rules (or predicates) and, specifying priority relationship between the rules using in-built "overrides" predicate. The scope of 'what is conflict' is specified by pair-wise mutual exclusion statements called "mutex's". For example: a mutex may specify that the grades of student can have only one value at one point of time. There is an implicit mutex between a rule (or predicate p) and its classical negation.

An illustration of Requirements Representation in Courteous Logic: Consider the scenario of book issue in a library. The requirements are often expressed with inconsistent and, possibly repetitive statements as we observed:

- If a person is a library member, then he can borrow the book.
- If a book is available, then library member can borrow the book.

These two statements of requirements specification are contradictory – the second statement adds one more condition for borrowing the book in addition to a person's being library member, namely: *availability of the book*. We have considered such a simple scenario to illustrate how minor mistakes in expressing the requirements can result in faulty software. One may consider that everyone knows about the library rules; however, it is not always possible that requirements analysts as well as test engineers are familiar with the domain knowledge under study. In the absence of formal specifications, requirements cannot be validated in the early phases of software development, nor an appropriate set of test-cases be generated. Ambiguity and inconsistency in requirements may continue to the test-cases as well. We have looked for solution to such a scenario in our earlier work. The above-mentioned requirements, when translated to courteous logic representations appear as:

```
<rule1> if  librarymember(?X) and
 book(?Y)
then    borrow(?X, ?Y) .

<rule2> if  librarymember(?X) and
 book(?Y) and status(?Y, available)
then    borrow(?X, ?Y) .
```

These two rules correspond to the two requirements statements stated above. Both of these rules are labeled as <rule1> and <rule2> respectively. Without any additional information, rule 1 may allow a book to be issued even if it is not available. This is contrary to real-world supposition that only an available book can be issued to a library member. The result of inference engine indicates that these requirements are inconsistent in nature and the

requirements are corrected in consultation with domain experts. The suggested correction to these requirements can be added as another rule to above courtoes logic representation of requirements as:

```
overrides(rule1, rule2).
```

Having obtained formal and consistent set of requirements specifications, we can design better test-cases; the process of which, we have automated in this work.

2.2 Related Work

Automated functional test generation has been an intriguing problem in research arena. Significant amount of effort towards test case generation has been reported in survey reports too like (Kaur and Vig, 2012), (Gutierrez et al., 2006). Kaur and Vig have attempted to find the most widely used UML diagrams for automated test case generation and the corresponding advantages. In addition to this, they have explored the type of testing is addressed and what are the challenges involved. Their findings indicate that activity diagrams, state diagrams and a combination of use-case diagram and activity diagram have been mostly used for generating test cases. They observe that functional testing has been the extensively studied; however, the challenges involved are incomplete or incorrect requirements specifications and UML diagrams. The survey conducted by Gutierrez et al. also suggests that UML models and use-case specifications have been mostly used for test case generation automatically.

Survey reports summarize that UML diagrams and requirements specifications in the form of use-cases (an NL representation) have been explored most for automated test case generation. Activity Diagrams have been used for the purpose in the works of (Boghdady et al., 2011), (Kansomkeat et al., 2010) and (Li et al., 2013). The fact that activity diagrams represent the behaviour of the real-world system for a given scenario can be attributed to the use of activity diagrams for test case generation. State charts form the basis of generation of test cases in the works of (Hartmann et al., 2005) and (Offutt and Abdurazik, 1999). Use-cases also describe the expected behaviour of the system. Therefore, use cases have also been used for test case generation as reported in the works of (Heumann, 2001) and (Ahlowalia, 2002). The user-stories used in agile development have also been considered for automatically generating test cases (Kamalkar et al., 2013). However as reported in the survey of Kaur and Vig, the challenge involved with these approaches is that of the representation of requirements. NL requirements representation results in ambiguity, incompleteness and inconsistency of requirements and consequently, generation of incorrect test cases.

However, there are few instances where test cases have been generated from either formal representation of requirements or using approaches or formal intermediate requirements representation like (Pretschner, 2001) and (Mandrioli et al., 1995). We also support formalism in requirements representation. In our work here, we have made use of courteous logic based representation of the requirements for automatically generating test cases.

3 OUR APPROACH

Our approach borrows from semantic-head-generation algorithm for unification-based formalisms as proposed by Sheiber et al., (1989). However, our goal is different from NL generation. NL generation has its own challenges. It requires "glue-word" in addition to the grammar rules followed for generating NL expressions from the source input (Grasso, 2000). Our interest lies only in generating the test-cases from courteous logic representations of requirements. Our courteous logic representations have been generated semi-automatically (only override predicates have been added manually), therefore the variable names and the predicate names are more meaningful and self-understood.

We first identify the pivot element for the input rule like Shieber et al.'s approach but we are not interested in considering it as semantic head unlike their approach. In our case, the pivot element is *predicate* or the *rule-head* of the given rule. For example: for the library rules discussed in section 2.1, the pivot element is '*borrow*'. Next we consider the body of the rule, which can simply be another predicate or clause (like rule-head) or conjunction of two or more predicates. We process each of these predicates one by one as described in the algorithm below. Test cases are laid out for null checks, invalid and valid values of the variables. NL generation is performed only for expressing the actual output. Since the actual outputs of test-cases are in terms of the pivot elements, which have been earlier generated from NL document only, we need not have to refer to any 'glue-words' in between. This reduces the complexity considerably in our approach. The algorithm for generating test-cases is shown in figure 1 below:

Algorithm: Test-case Generation from Courteous Logic Representation of Requirements

Input: Set of requirements represented in the form of courteous logic

Output: Set of test-cases for the input requirements

Processing Steps:

1. For each rule in the input file (collection of rule-base):
2. Extract the label (optional) and the pivot predicate (identified as predicate after 'then') and store them for future reference.
3. Start generating test-cases for the rule-body:
4. If there is a single predicate in rule-body,
   (i) Then, for each variable (recognised by a prefix character '?'), add test-cases for Nullness check and validity check. Express actual result as "Error Message displayed" for null and invalid values.
   (ii) Else, store each of the predicates joined by conjunctions separately. Process each of these predicates as described in the step (i)
5. If the label encountered has higher precedence in any of the 'override' predicates, then
   (i) Search for the stored pivots and ensure that conditions for both pivots corresponding to the labels in 'override' hold good.
   (ii) Add test-case stating: "Enter valid values for variables such that both the <pivots> hold good".
   (iii) Actual result in this case should be expressed as: "The results obtained after executing the test case should correspond to <prioritized pivot >".

Note: The pivots expressed in angular brackets <> are to be replaced by their actual values for the rule under process.

Figure 1: Algorithm for test-case generation.

4 CASE STUDY

We conducted our case-study for test-case generation on requirements from various business domains like banking, academics and health-insurance. In this section, we will consider same examples as illustrated in our earlier work (Sharma and Biswas, 2012) so that establishing the relationship between the requirements studied and the generated test-cases will be easy. With this current work, we have modified the previous algorithm for generating the courteous logic representations to generate predicates and variable with complete words instead of mnemonics. This modification has been done to reduce number of look-ups in mnemonics database for test-case generation. Therefore, the representations of requirements illustrated in following sub-sections will slightly differ in having complete words instead of mnemonics.

4.1 Case Study – I

***Example 1* - Representing and Prioritizing Conflicting Views (Academic Grade Processing):** This example is about the specifications of students' grade approval process where the students' grades are approved by the course-coordinator, the department head and the dean. The expected behaviour of the system refers to the fact that at any point in time, approval from department head holds higher priority over course-coordinator; and approval from dean higher priority over department head and in turn, the course coordinator. In order to capture this observable behaviour, we have earlier suggested the use of courteous logic representations as shown below:

```
<new>
   if assignGrades (
   ?RegistrationNumber,   ?Year,
   ?Semester,  ?Group, ?Subject,
   ?GradePoint )
   then value_Status   (
   new, ?RegistrationNumber,  ?Year,
   ?Semester,  ?Group, ?Subject);
<cdn>
   if approvedby       (
   ?RegistrationNumber,   ?Year
   ?Semester,  ?Group, ?Subject, ?Point,
   ?Status,    coordinator )

   then value_Status   (
   coordinatorApproved,
   ?RegistrationNumber,   ?Year,
   ?Semester,  ?Group, ?Subject );

<hod>
   if approvedby       (

   ?RegistrationNumber,   ?Year,
   ?Semester,  ?Group, ?Subject, ?Point,
   coordApproved,  hod )
```

```
   then value_Status   (
   hodApproved, ?RegistrationNumber,
   ?Year,   ?Semester, ?Group,
   ?Subject );

<dean>
   if  approvedby  (
   ?RegistrationNumber,   ?Year,
   ?Semester,  ?Group, ?Subject,
   ?Point, hodApproved, dean)

   then value_Status   (
   deanApproved, ?RegistrationNumber,
   ?Year,   ?Semester, ?Group,
   ?Subject );

overrides(cdn, new);
overrides(hod, new);
overrides(dean, new);
overrides(hod, cdn);
overrides(dean, cdn);
overrides(dean, hod);
MUTEX
      value_Status(?Status1,
      ?RegistrationNumber, ?Year,
      ?Semester, ?Group, Subject  )
      AND
      value_Status(    ?Status2,
      ?RegistrationNumber,         ?Year,
      ?Semester, ?Group, Subject  )
GIVEN notEquals( ?Status1, ?Status2 );
```

The test cases generated corresponding to above requirements for nullness and validity checks and then for functional test-cases have been presented in table 1 below:

4.2 Case Study - II

***Example 2*– Representing Default and Exceptional Scenario Processing (Saving and Current Account Processing):** Consider the account processing part of a bank customer where he can have more than one account. Let's consider that a bank customer can have a current account and a saving account. The customer can choose one of these accounts as default account for any transaction that he wants to carry out. The usual choice is current account but to keep the use-case generic, let us assume that customer has marked one of the accounts as default. The customer is free to select the other account for some of his transactions. In that case, the selected account processing should override the default processing.

Table 1: Test Cases – Case Study I.

Sl. No.	TEST CASE	ACTION PERFORMED	ACTUAL RESULT
1.	Null Checks for 'assignGrades'	Enter null value of RegistrationNumber	Error Message displayed
2.		Enter null value of Year	Error Message displayed
3.		Enter null value of Semester	Error Message displayed
4.		Enter null value of GradePoint	Error Message displayed
5.	Validity Checks for 'assignGrades'	Enter invalid value of RegistrationNumber	Error Message displayed
6.		Enter invalid value of Year	Error Message displayed
7.		Enter invalid value of GradePoint	Error Message displayed
8.	Execute assignGrades	Enter valid values for variables: RegistrationNumber, Year, Semester, Group, Subject, GradePoint	Value of grade status is 'new'
9.	Execute approvedby (under label – cdn)	Enter valid values for variables such that both the 'assignGrades' and 'approvedby' hold good.	The results obtained after executing the test case should correspond to 'approvedby'. Value of grade status is 'coordinatorApproved'

Table 2: Test Cases – Case Study II.

Sl. No.	TEST CASE	ACTION PERFORMED	ACTUAL RESULT
1.	Null Checks for 'deposit' (Similarly for withdraw)	Enter null value of TransactionId	Error Message displayed
2.		Enter null value of AccountId	Error Message displayed
3.		Enter null value of Amount	Error Message displayed
4.	Validity Checks for 'deposit'	Enter invalid value of TransactionId	Error Message displayed
5.		Enter invalid value of AccountId	Error Message displayed
6.	Execute add_Amount	Enter valid values of TransactionId , Client, Amount and AccountId	Add amount – Default Account
7.	Execute add_Amount	Enter valid values of TransactionId , Client, Amount and AccountId and option 'select' provided	Amount added to select Account

The natural language expression for such default operation and associated exception can be easily understood by the involved stakeholders as well as developers. But what is often overlooked by developers is the implicit interpretation here – the account chosen for default processing should remain unaffected in case selection is made for the non-default account and often, this is uncovered till testing phase. Such overlooked implicit interpretation results in implicit internal inconsistency. Such a defect can be easily detected during RE phase if we have an executable model for representation of requirements that can sufficiently express the domain knowledge. We have translated the requirements for this scenario in courteous logic from NL as:

```
<def>
 if deposit(?TransactionId, ?Client,
 ?Amount) and
 holds(?Client, ?AccountId)  and
 default(?AccountId)
 then add_Amount(?Client, ?AccountId,
 ?Amount);
<sel>
 if deposit(?TransactionId, ?Client,
 ?Amount) and
 holds(?Client, ?AccountId) and
 option(?Client, ?TransactionId,
 select, ?AccountId)

 then add_Amount(?Client, ?AccountId,
 ?Amount);
<def>
 if withdraw(?TransactionId, ?Client,
 ?Amount) and
 holds(?Client,    ?AccountId)    and
 default(?AccountId)
 then subtract_Amount(  ?Client,
 ?AccountId, ?Amount);

 <sel>
 if withdraw(?TransactionId, ?Client,
 ?Amount) and
 holds(?Client, ?AccountId) and
 option(?Client, ?TransactionId,
 select, ?AccountId)
 then subtract_Amount(  ?Client,
 ?AccountId, ?Amount);

overrides(sel, def);
MUTEX
 add_Amount(?Client,    ?Account1,
 ?Amount)    AND
 add_Amount(?Client,    ?Account2,
 ?Amount)
GIVEN notEquals(?Account1,?Account2)

MUTEX
 subtract_Amount(?Client,  ?Account1,
 ?Amount) AND
 subtract_Amount(?Client,  ?Account2,
 ?Amount)
GIVEN notEquals(?Account1,?Account2)
```

The test cases generated corresponding to second case-study have been presented in table 2 above. We performed case-study on various other scenarios from our requirements corpus and have found satisfactory results. To check validity of our generated test-cases, we compared our test-cases against the corresponding system test-cases and found that our test cases were actually a superset of otherwise manually written test-cases. We also checked the usability of our test-cases by executing

these test-cases on the system under study. The first authors of this paper carried out this usability check and the author did not find it difficult to comprehend these automatically generated test-cases.

5 DISCUSSION AND CONCLUSION

In this paper, we have presented an approach to automatically generate functional test cases from courteous logic representation of the requirements. The approach borrows from semantic head-driven approach for NL Generation proposed by Shieber et al. The advantage of our approach is that courteous logic representations have English-like constructs and easy to process. Secondly, we are generating these representations from NL requirements, therefore the courteous rules representing requirements become self-explanatory and with limited set of support words, we have been able to generate the functional test cases automatically. We are interested in validation of our test-cases by test-engineers themselves. We further plan to improve our algorithm with the feedback obtained and design a tool supporting our approach.

REFERENCES

Ammann, P. and Offutt, J., 2008, *Introduction to Software Testing*, Cambridge University Press, USA.

Incomplete List of Testing Tools. Available from: http://research.cs.queensu.ca/~shepard/testing.dir/under.construction/tool_list.html. [20 January 2014].

Tsai, J.J.P. and Weigert, T., 1991, HCLIE: a logic based requirement language for new Software Engineering Paradigms, *Software Engineering,* vol. 6, no. 4, pp. 137-151.

Boghdady, P.N., Badr, N.L., Hashem, M. and Tolba, M.F., 2011, A Proposed Test Case Generation Technique Based on Activity Diagrams, *International Journal of Engineering and Technology,* vol. 11, no. 3, pp. 35-52.

Kansomkeat, S., Thiket, P. and Offutt, J., 2010, Generating Test Cases from UML Activity Diagrams using the Condition Classifcation Method, In 2nd *International Conference on Software technology and Engineering (ICSTE'10),* San Juan, pp. V1-62 – V1-66.

Li, L., Li, X., He T. and Xiong, J., 2013, Extenics-based Test Case Generation for UML Activity Diagram, In *International Conference on Information Technology and Quantitative Management (ITQM'13)*, pp. 1186-1193.

Heumann, J., 2001, Generating Test Cases from Use Cases, In *the Rational Edge, e-zine for Rational Community*.

Ahlowalia, N., 2002, Testing from Use Cases Using Path Analysis Technqiue, Analysis, In *International Conference on Software Testing Analysis and Review*.

Kamalkar, S., Edward, S.H. and Dao, T.M., 2013, Automatically Generating Tests from Natural Language Descriptions of Software Behavior, In *8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'13)*.

Sharma R. and Biswas, K.K., 2012, A Semi-automated Approach towards Handling Inconsistencies in Software Requirements. In: Maciaszek, L.A. and Filipe, J. (eds.), *Evaluation of Novel Approaches to Software Engineering,* Springer Berlin Heidelberg, pp. 142-156.

Sharma, R. and Biswas, K.K. 2012, Using Norm Analysis Patterns for Requirements Validation, In *IEEE 2nd International Workshop on Requirements Patterns (RePa)*, pp.23-28.

Shieber, S.M., Noord, G.N, Moore R. and Pereira, C.N., 1989, A Semantic-head-driven generation algorithm for unification-based formalisms, In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics,* pp. 7-17.

Reiter, R., 1980, A logic for default reasoning, Artificial Intlligence, vol. 13, pp. 81-132.

Nute, D., 2001, Defeasible Logic, In *Proceedings of International Conference on Applications of Prolog (INAP 2001)*, IF Computer Japan, 2001, pp 87-114.

Grosof, B.N., 1997. Courteous Logic Programs: prioritized conflict handling for rules. *IBM Research Report RC20836, IBM Research Division, T.J. Watson Research Centre.*

Grosof, B.N., 2004. Representing E-Commerce Rules via situated courteous logic programs in RuleML. *Electronic Commerce Research and Applications*, Vol 3, Issue 1, 2004, pp 2-20.

Kaur, A. and Vig, V., 2012, Systematic Review of Automatic Test Case Generation by UML Diagrams, *International Journal of Engineering Research and Technology*, vol. 1, no. 7.

Gutierrez, J.J., Escalona, M.J., Mejias, M. and Torres, J., 2006, Generation of test cases from functional requirements. A survey, In *4th workshop on System Testing and Validation,* Potsdam, Germany.

Hartmann, J., Vieira, M., Foster, H., Ruder, A., 2005, A UML-based Approach to System Testing, *Journal of Innovations System Software Engineering*, vol. 1, pp. 12-24.

Offutt, J. and Abdurazik, A., 1999, Generating Tests from UML Specifications, In *UML'99 – The Unified Modeling Language*, Springer Berlin Heidelberg, pp. 416 – 429.

Pretschner, A., 2001, Classical search strategies for test case generation with Constraint Logic Programming, In *Proceedings of International Workshop of Formal Approaches to Software Testing of Software,* pp. 47-60.

Mandrioli, D., Morasca, S. and Morzenti, A., 1995, Generating Test Cases for Real-Time Systems from Logical Specifications, *ACM Transactions on Computer Systems,* vol. 13, no. 4, pp. 365-398.

Grasso, F., 2000, Natural Language Processing: many questions, no answers, Available from: http://www.academia.edu/2824428/Natural_Language_Processing_many_questions_no answers. [20 January, 2014]