AUTOMATED REQUIREMENTS MODELLING WITH ADV-EARS

D. Majumdar[1] S. Sengupta[2] A. Kanjilal[3] S. Bhattacharya[4]

[1]RCC Institute of Information Technology, Kolkata – 700015, India

dipankar.majumdar@gmail.com

[2,3]B.P. Poddar Instt. of Mgmt. and Tech, Kolkata – 700052, India

sabnam_sg@yahoo.com

[4]Jadavpur University, Kolkata – 700032, India

bswapan2000@yahoo.co.in

ABSTRACT

The development of complex systems frequently involves extensive work to elicit, document and review functional requirements that are usually written in unconstrained natural language, which is inherently imprecise. Use of Formal techniques in Requirement Engineering would be of immense importance as it would provide automated support in deriving use case models from the functional requirements. In this paper we propose a formal syntax for requirements called Adv-EARS. We define a grammar for this syntax such that a requirements document in this format can be grammatically parsed and the prospective actors and use cases are automatically derived from the parse tree. The use case diagram is then automatically generated based on the actors and use cases and their relationships. We have used requirements of an Insurance system as a case study to illustrate our approach.

KEYWORDS

Requirements-Engineering, Adv-EARS, Automated Use case derivation, Formal requirements syntax.

1. INTRODUCTION

Object Oriented Programming is presently the most widely accepted programming paradigm and UML has become the de-facto standard for modeling an information system based on a given set of functional requirements of an Object Oriented system. However, in most of the cases the functional requirements expressed in natural language needs to be interpreted manually and converted to UML Use case diagrams. Such manual conversion often leads to missing information or incorrect understanding of the user needs. In this paper we define a formal syntax for requirements in a manner similar to natural language such that it is easy to use and understand. Mapping of the grammatical constructs of English has been done in our model named Adv-EARS [19]. This is built upon an earlier work by Marvin et al in [1] who have defined the Easy Approach to Requirements Syntax (EARS). Our model extends a lot on the basic four constructs and gives the flexibility is expressing any type of requirements. A grammar is then proposed which would parse the Requirements documents written in Adv-EARS and identify the elements of use case diagram from which the latter would be generated automatically.

This would, on one hand, greatly reduce ambiguities and errors in manual conversions. On the other hand, this will ensure a consistent evolution of use case models from requirements which forms the starting point for analysis and design models. Automated support would minimize early errors and provide a foundation for good quality software.

DOI : 10.5121/ijitcs.2011.1406

2. RELATED WORK

The objective of our work is to provide automated support for unambiguous interpretation of functional requirements and derivation of Use case models from those requirements. This is only possible through the use of formal techniques. Although there are several proposals to transform a more formal representation into use cases diagrams. For instance, (Stolfa and Radecký, 2004) and (Dijkamn and Joosten, 2002) transforms UML activity diagrams into use case diagrams, and (van Lamsweerde, 2009) transforms goal diagrams into use case diagrams. Formalizations of Textual Requirements to UML Diagrams is however scarce. Hence we here review the research works in the domain of formal specification of requirements.

Mavin et al in [1] put forward the there are three common forms of ambiguity in requirement specification: lexical, referential and syntactical [2]. To overcome such problems that arise because of the association with Natural Language (NL), usage of other notations has been advocated for the specification of user requirements. Z [3], Petri Nets [4] and graphical notations such as Unified Modeling Language (UML) [5, 6] and Systems Modeling Language (SysML) [7] are worth mentioning in this domain of work.

There are also numerous scenario-based approaches [8], tabular approaches such as Table-Driven Requirements [9] and ConCERT [10, 11] and pseudocode. However, use of any of these non-textual notations often requires complex translation of the source requirements, which can introduce further errors. There are also many research works specifically about how to write better requirements like [12, 13] that focus on the characteristics of well-formed requirements and the attributes that should be included. There are also templates available, such as VOLERE [14] and SL-07[15]. Despite this large body of research works, there seems to be little simple, practical advice for the practitioner. A set of simple requirement structures would be an efficient and practical way to enhance the writing of high-level requirements. Previous work in the area of constrained natural language includes Simplified Technical English [16], Attempt to Controlled English (ACE) [17], Easy Approach to Requirements Syntax (EARS) in [1] and Event-Condition-Action (ECA) [18]. In ECA, the event specifies the signal that triggers the rule and the condition is a logical test that (if satisfied) causes the specified system action.

We have extended the EARS framework proposed in [1] and proposed Adv-EARS, which is an advanced version of EARS for specifying requirement syntax, to incorporate a few more types of requirements and proposed a context-free-grammar for these requirements that is parsed to generate the use case diagram. The Adv-EARS syntax is much better suited and more generically defined to handle most kinds of functional requirements.

3. SCOPE OF THE WORK

In this paper we propose a framework for expressing requirements in a formal syntax named Adv-EARS [19] which is parsed by a grammar to identify potential use cases and actors. Our approach then enables automatic derivation of Use case diagrams based on the actors, use cases and their relationships identified from the parser. In addition to our previous work [19], the current paper shows and demonstrates the complete Context Free Grammar (CFG). The paper also shows the Parse Trees produced when a parser, generated from the said CFG is applied to the Requirements of an Insurance System for parsing.

Use Case identification is a customary procedure in the Requirements Engineering phase, where the potential use-cases are identified along with their intrinsic behavioral pattern(s). These Use-cases are the manifestations of the user requirements at the later part of the Requirements-Engineering process. This automated approach of aspect identification during early part of requirement-engineering phase will be of significant importance. In this paper, we represent the requirements in natural language following a formal syntax as defined in EARS [1]. However, the

EARS syntax is not sufficient to enable automatic derivation of use cases. Therefore, we have further extended EARS [1] to include some more constructs to handle the different kinds of requirements and name this as Adv-EARS. We present a Context Free Grammar for the Adv-EARS. A parser, designed based on the CFG yields a parse tree. The leaf nodes of the parse tree highlight the probable use cases along with the Use-Case Relationships. Consequently, with optional or minor discretionary intervention of the designer, the Use Case diagram can be generated automatically.

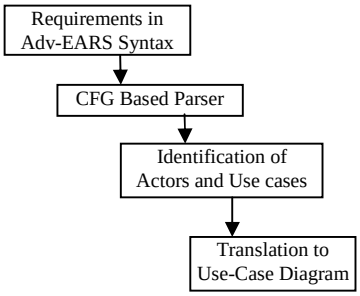

Fig 1: Diagrammatic representation of our approach

Fig 1 shows the diagrammatic representation of our approach. Textual requirements are represented in a formal language based on Adv-EARS. This structured requirement is parsed using the grammar defined and the parse tree identifies the probable actors, use cases and use case relationships. This information is used to automatically derive the Use case diagram for the system.

4. DERIVATION OF USE-CASE DIAGRAM FROM REQUIREMENT

The automatic derivation of use case diagram from requirements expressed in formal syntax comprises of three steps as shown in Fig 1. In the 1st section, we first formally define the software requirement using Adv-EARS model. In the 2nd section, a CFG grammar is defined for parsing the requirements expressed in Adv-EARS. The actors and use cases are identified from the generated parse tree. Finally in the 3rd section we derive the use case diagram from the identified actors and use cases and their relationships.

5. THE ADV-EARS MODEL

For the sake of automating the process of Requirements Engineering, we adopt the EARS [1] based Requirements Syntax. The EARS syntax has placed the Requirements under various classification heads: Ubiquitous, Unwanted Behavior, Event Driven and State Driven. The principal objective of this paper is to decompose the phrases identified in EARS [1] further with an aim to identify the involved Entities and Use Cases. Furthermore, the current paper extends the list and thereby adds a new head namely the Hybrid Requirement that is a combination of Event-Driven and Conditional. This is triggered by an event but at the same time having a condition for its execution. This is to take care of requirements that may be of this nature. Moreover the definition of EARS[1] for the existing categories have been also extended and generalized by adding few more different types of syntaxes so that Adv-EARS becomes better suited for use for formal requirement definition.

We use the following codes for the different types of requirements syntax, which are defined in the Table-1 as given below -

UB: Ubiquitous, EV: Event-driven, UW: Unwanted Behavior, ST: State driven, OP: Optional features, HY: Hybrid (Event-Driven and Conditional)

Table-1
Requirement Types of Adv-EARS and difference in definition from EARS

Req Type	Definition in EARS	Definition in Adv-EARS (extensions in bold)
UB	The <system name> shall <system response>	The <entity> shall <functionality> | **The <entity> shall <functionality> the <entity> for <functionality>**
EV	WHEN <optional preconditions> <trigger> the <system name> shall <system response>	When <optional preconditions> the <entity> shall <functionality> | **When <optional preconditions> the <entity> shall perform <functionality> | When <entity> <functionality> the <entity> shall <functionality>**
UW	IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>	IF < preconditions> THEN the <entity> shall <functionality> | **IF < preconditions> THEN the <functionality> of <functionality> shall <functionality> | IF < preconditions> THEN the <functionality> of <functionality> shall <functionality> to <functionality> | IF < preconditions> THEN the <functionality> of <functionality> shall <functionality> to <functionality> and <functionality>**
ST	WHILE <in a specific state> the <system name> shall <system response>	WHILE <in a specific state> the <entity> shall <functionality> | **WHILE <in a specific state> the <functionality> shall <functionality>**
OP	WHERE <feature is included> the <system name> shall <system response>	WHERE <feature is included> the <entity>shall <functionality> | **WHERE < preconditions> the <functionality> shall <functionality> | WHERE < preconditions> the <functionality> of <functionality> shall <functionality> to <functionality>**
HY	Not defined	**<While-in-a-specific-state> if necessary the <functionality> shall <functionality> | <While-in-a-specific-state> if necessary the <entity> shall perform <functionality> | <While-in-a-specific-state> if <preconditions> the <functionality> shall <functionality>**

Summarizing, the contribution of Adv-EARS–

1. Introduction of a new type of Requirement Hybrid (HY) which is an event driven and conditional requirement.
2. Extension of all existing Requirement syntaxes to make it more generic and able to handle more different types of requirements.
3. Instead of <system name> we use <entity> which corresponds to the entities (external & internal) interacting with the software system. These are the nouns and some of them maps to possible actors of the system.
4. Instead of <system response> we use <functionality> which corresponds to the use cases of the software system. These are the verbs and some of them maps to the use cases of the system.

Assumption: We are assuming that there is one statement per requirement that may be simple or complex. If complex, there may be more than one related requirements. The use cases that are

derived by parsing these complex requirements expressed in Adv-EARS format, are also related. A single instance a parse tree having more than one use cases as its leaf nodes indicates the relationship among those use cases. These relationships among one or more use cases along with the use cases determine the Use-case diagram. In this manner, we propose to automate the generation of Use Case diagram from Adv-EARS form of requirement.

6. ADV-EARS BASED CONTEXT FREE GRAMMAR

We have defined a CFG (Context Free Grammar) for the requirements formatted in Adv-EARS syntax. Our CFG, as shown in Fig 2, when implemented using 'lex' and 'yacc' programs available with Red Hat Enterprise Linux 5.0 generates a parse tree at its runtime. Such a parse tree unveils the potential use-cases of the system and their inter-relationships.

```
%token The the shall as a an word When operation
% token trigger then While Where
%start translation_unit
%%
translation_unit: requirement_def
                | translation_unit requirement_def;
requirement_def:
        The <entity> shall <functionality>
        | The <entity> shall <functionality> the <entity> for <functionality>
    | When <preconditions> the <entity> shall <functionality>
    | When <preconditions> the <entity> shall perform <functionality>
    | When the <entity> shall <functionality>
        | When <entity> <functionality> the <entity> shall <functionality>
        | IF <preconditions> THEN the <entity> shall <functionality>
    | IF <preconditions> THEN the <functionality> of <functionality> shall <functionality>
        | IF <preconditions> THEN the <functionality> of <functionality> shall <functionality> to <functionality>
        | IF <preconditions> THEN the <functionality> of <functionality> shall <functionality> to <functionality> and <functionality>
      | WHILE <in a specific state> the <entity> shall <functionality>
    | WHILE <in a specific state> the <functionality> shall <functionality>
    | WHERE <feature is included> the <entity>shall <functionality>
    | WHERE <preconditions> the <functionality> shall <functionality>
    | WHERE <preconditions> the <functionality> of <functionality> shall <functionality> to <functionality>
| <While-in-a-specific-state> if necessary the <functionality> shall <functionality>
| <While-in-a-specific-state> if <preconditions> the <functionality> shall <functionality>
text: word | text word;
entity: word;
system: text;
functionality: text;
remainder-text: text;
preconditions: text;
in-a-specific-state: text;
feature-is-included: text;
%%
```

Fig-2: Context Free Grammar for Adv-EARS

7.GENERATION OF USE CASE DIAGRAM FROM PARSE TREE

The parse tree yielded by the CFG as mentioned in the previous section generates the Use Cases at specific points of its leaf nodes as terminal symbols. The probable entities and Use Cases, which appear as leaf nodes of the parse tree distinctly are identified. A single instance a parse tree having more than one Use Cases as its leaf nodes indicates the relationship among those Use Cases. This relationships among one or more Use-Cases determine the Use-case diagram.

As a result of which the generation of Use-case diagram becomes entirely mechanized and can highly be automated with minor intervention of the designer.

The key points are:

1. The <functionality> parts of Adv-EARS form the probable use cases of use case diagram.
2. The <entity> parts of Adv-EARS form the actors and/or classes of the system. Minor manual intervention is required to choose the actors of the use case diagram
3. The <functionality> shall <functionality> parts of Adv-EARS form the "includes" relationship of the 1st use case to the 2nd use case.
4. The <functionality> of <functionality> also corresponds to use case relationship. The relationship "includes" or "extends" is decided by the designer.

For example, for a requirement like "If <age is less than 18>, then *perform validation process* shall *invoke error-handler* to reject voter identity application". As per the requirement definition of UW (Unwanted behavior) (3rd case) from Table-1, we can derive the probable actors and use cases corresponding to <entity> and <functionality>

- Use cases: "perform validation process", "invoke error-handler", "reject voter identity application "
- Use case "perform validation process" holds the <<includes>> relationship with the "invoke error-handler" use case.

Similarly, for the requirement "The *user* shall *login*",

As per the definition of Ubiquitous requirements (UB) in Table-1, we can derive –

- Actor: Applicant, Insurance-Officer
- Use Case: login

The "extends" relationship among the use cases is not within the scope of the use case diagram generated from the parse trees derived from requirements expressed in Adv-EARS format by using a context free grammar. Designer intervention is required to change "includes" to "extends" if required. The generated use case diagram would be a starting point which designers can use as a template to create better designs.

8.CASE-STUDY

We have taken a case study of insurance system to illustrate our approach. This is a common Insurance System followed in most of the Insurance Companies for general and life insurance plans. The following list presents the Requirements Document in Textual form of Natural Language. These are to be formulated according to the EARS Requirements Syntax as shown in earlier section.

Requirements for Insurance System:

a. Applicant logs in to the system
b. Applicant wants to apply for an Insurance Policy
c. Applicant selects an insurance product
d. The policy application form accepts all details from Applicant

e. Underwriting1 performed to validate Applicant details based on product rules
f. Some sample validations –
    i. If age > age_limit years, application is rejected
    ii. If beneficiary is not related to policyholder, application is rejected
    iii. If profession is risky, application rejected
    iv. If smoker, set premium value high
    v. If user holds previous policies, then policylimit is verified (where a Applicant cannot have policies whose total policy amount exceeds a limit)
g. After underwriting1, if required further details are accepted from Applicant
h. Underwriting2 is performed after new details are obtained
i. Premium is calculated based on product choice and Applicant details
j. Premium payment is accepted
k. If payment is done, policy is created
l. Policy Certificate is generated and issued

8.1 Requirements in Adv-EARS Format

The Requirements Syntax for the Insurance System as presented in Sec 4.1 is reformulated according to the EARS Syntax is presented in the Table-2.

Table-2: Advanced EARS Syntax for Insurance System

Sl. No.	Requirements	Type
1	The *applicant* shall *login*	UB
2	If applicant fails to login then the *login functionality* shall *invoke error-handler*	UW
3	The *applicant* shall *select product details*	UB
4	When product selected the *applicant* shall *provide application details*	EV
5	The System shall authenticate for User Login	UB
6	When application details have been accepted the *Insurance Officer* shall perform *underwriting1*	EV
7	While underwriting1 is performed the *error-handler* shall *handle exceptions*	ST
8	If age is greater than age-limit then the *Checking* of *Underwriting1* shall *invoke Error-Handler* to *reject the application*	UW
9	If profession is risky, then the *Checking* of *Underwriting1* shall *invoke Error-Handler* to *reject the application*	UW
10	If beneficiary is not related to policyholder, then the *Checking* of *Underwriting1* shall *invoke Error-Handler* to *reject the application*	UW
11	Where the beneficiary is a smoker the *Checking* of *Underwriting1* shall *invoke Error Handler* to *set a high value premium.*	OP

Sl. No.	Requirements	Type
12	After underwriting1 is complete, if necessary the *Insurance Officer* shall perform *underwriting2*	HY
13	When underwriting2 is required the *applicant* shall *provide more information*	EV
14	While underwriting2 is performed the *invoke error-handler* shall *handle- exceptions*	ST
15	When underwriting1 and underwriting2 is completed error free the *Insurance Officer* shall *calculate premium*	EV
16	When premium has been calculated the *applicant* shall *make premium payment*	EV
17	If payment fails, the payment-functionality shall *invoke error-handler* to *abort transaction* and *notify user*	UW
18	When payment is done the *Insurance Officer* shall *create policy*	EV
19	When policy issued the *Insurance Officer* shall *generate policy certificate*	EV

The above Adv-EARS syntax elements, when fed to the parser, it generates the following Parse Trees. For the sake of brevity of space, we have shown one of each type instead of all the types of syntaxes.

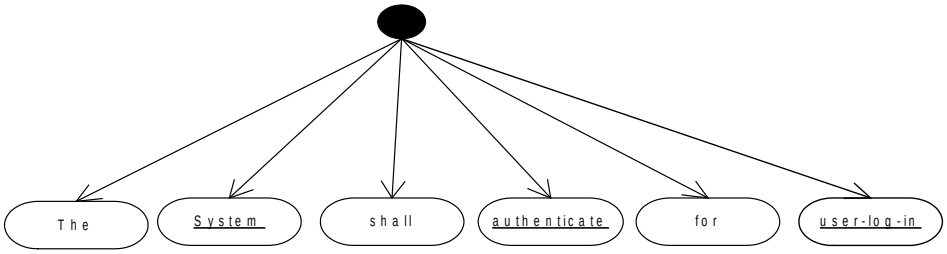

Fig 3(a): Parse Tree for the Syntax No: 5

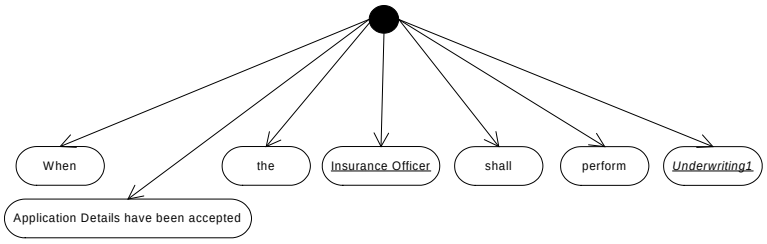

Fig 3(b): Parse Tree for the Syntax No: 6

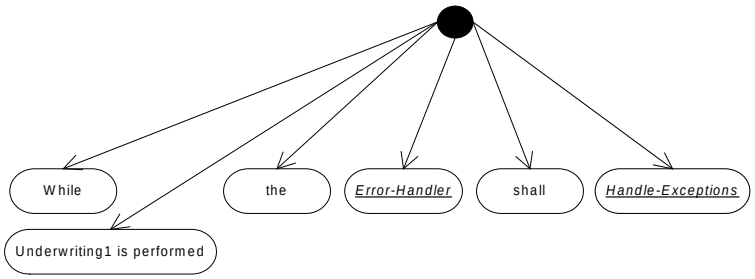

Fig 3(c): Parse Tree for the Syntax No: 7

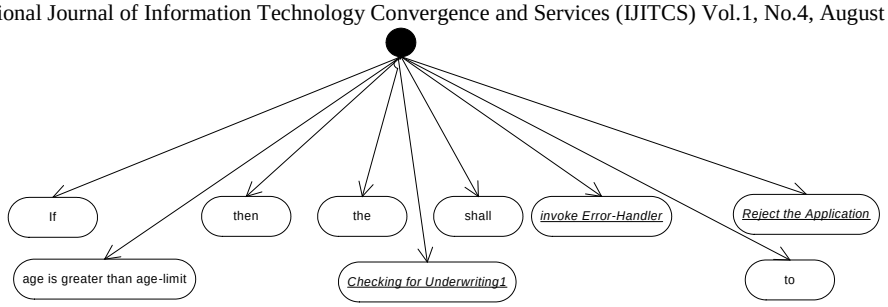

Fig 3(c): Parse Tree for the Syntax No: 8

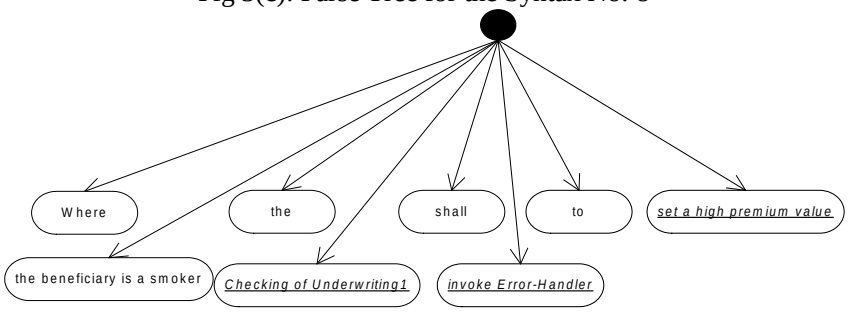

Fig 3(d): Parse Tree for the Syntax No: 11

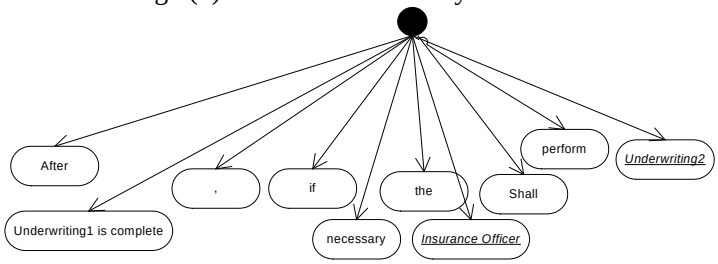

Fig 3(e): Parse Tree for the Syntax No: 12

8.2 PARSE TREE TO USE CASE DIAGRAM

The above document when fed to the Context Free Grammar presented in Fig-2 for the Ad vanced EARS Syntax generates a parse tree shown as Fig 3(a) through 3(e). The Use Case Diagram, derived from the parse trees is shown in Fig 4.

9. DISCUSSION AND CONCLUSION

Use of Formal techniques in Requirement Engineering would be of immense importance as it would provide automated support in deriving use case models from the functional requirements. In this paper we propose a formal syntax for requirements called Adv-EARS.

We define a grammar for this syntax such that a requirements document in this format is grammatically parsed and the prospective actors and use cases are automatically derived from the parse tree based on a Context Free Grammar for Adv-EARS. The use case diagram is then automatically generated based on the actors and use cases and their relationships. The Adv-EARS is still extendible and our future work will be to consolidate and present all possible require-

ments expressed in natural language in Adv-EARS format. Once the Adv-EARS is able to map to most of the constructs of English grammar in a controlled manner, it would take us steps forward in Automated Requirement Engineering, lowering manual intervention and hence lowering the probability of human errors and ambiguity.

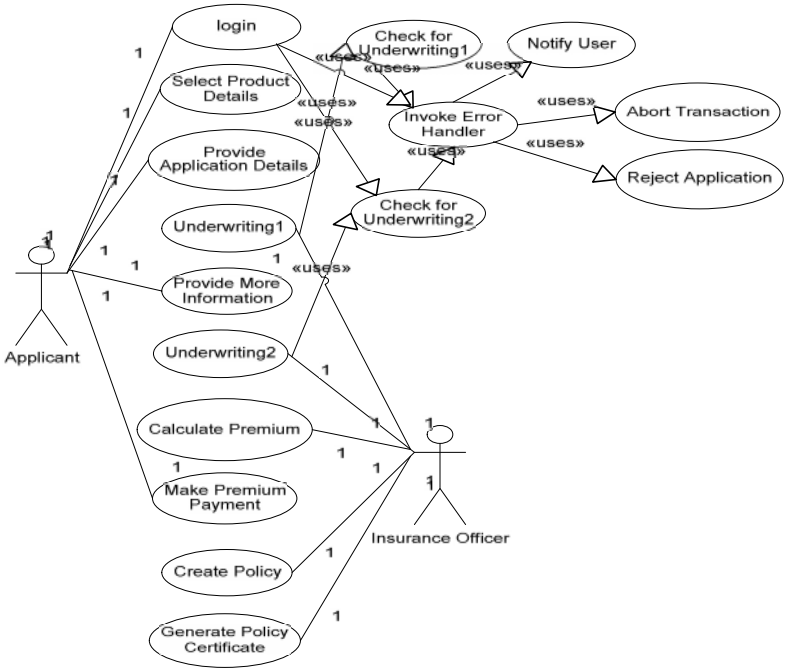

Fig-4: Use-Case Diagram for Insurance System

10. REFERENCES

[1] Alistair Mavin, Philip Wilkinson, Adrian Harwood, Mark Novak, Easy Approach to Requirements Syntax (EARS), 2009 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31-September 04

[2] Warburton, N., "Thinking from A to Z", 2nd edition, Routledge, 2000.

[3] Woodcock, J. and Davies, J., "Using Z-Specification, Refinement and Proof", Prentice Hall, 1996.

[4] Peterson, J. "Petri Nets", ACM Computing Surveys 9 (1977), 223-252.

[5] Object Management Group, UML Resource Page, http://www.uml.org/

[6] Holt, J., "UML for Systems Engineering: Watching the Wheels" (2nd edition), IEE, 2004.

[7] Object Management Group, Official OMG SysML Site, http://www.omgsysml.org/

[8] Alexander, I.F. & Maiden, N.A.M. (eds), "Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle" Wiley, 2004.

[9] Alexander, I.F. & Beus-Dukic, L., "Discovering Requirements",John Wiley, 2009.

[10] Vickers, A., Tongue, P.J. and Smith, J.E., "Complexity and its Management in Requirements Engineering", INCOSE UK Annual Symposium – Getting to Grips with Complexity, Coventry, UK, 1996.

[11] Vickers, A., Smith, J.E., Tongue, P.J. and Lam, W., "The ConCERT Approach to Requirements Specification (version 2.0)", YUTC/TR/96/01, University of York, November 1996 (enquiries about this report should be addressed to: High-Integrity Systems Engineering Research Group, Department Computer Science, University of York, Heslington, YORK, YO10 5DD, UK).

[12] Hooks, I., "Writing Good Requirements", Proceedings of Third International Symposium of INCOSE Volume 2, INCOSE, 1993. [12]

[13] Wiegers, K., "Writing Good Requirements", Software Development Magazine, May 1999.

[14] "VOLERE Requirements Specification Template", Atlantic Systems Guild, ttp://www.volere.co.uk/template.htm

[15] Lauesen, S., “Guide to Requirements SL-07. Template with Examples”, Lauesen Publishing, 2007.

[16] “ASD Simplified Technical English: Specification ASD-STE100. International specification for the preparation of maintenance documentation in a controlled language”, Simplified Technical English Maintenance Group (STEMG), 2005.

[17] Fuchs, N. E., Kaljurand, K. and Schneider, G., "Attempto Controlled English Meets the Challenges of knowledge Representation, Reasoning, Interoperability and User Interfaces", FLAIRS, 2006.

[18] Dittrich, K. R., Gatziu, S. and Geppert, A., “The Active Database Management System Manifesto: A Rulebase of ADBMS Features.”, Lecture Notes in Computer Science 985, Springer, 1995, pages 3-20.

[19] Dipankar Majumdar, Sabnam Sengupta, Ananya Kanjilal, Swapan Bhattacharya, “Adv-EARS: A Formal Requirements Syntax for Derivation of Use Case Models”, Proc of the First International Conference on Advances in Computing and Information Technology, July, 15-17, Chennai, India. pp 40-48.