Use of Finite State Automaton as an attempt to Simplify, Automate and Improve Requirements Engineering

AMIT SUGANDHI
Tata Consultancy Services Ltd.
Everett, Washington
UNITED STATES OF AMERICA
amit.sugandhi@tcs.com

Abstract: - Requirements Engineering is the phase from where a Software System begin it’s life. Requirements gathered and analyzed in this phase is designed, coded, tested and finally delivered in the form of a Software. Incomplete, misinterpreted or missed requirements may cause budget over run, schedule slippage and a not so good product at the end. Thus, it is really important to derive as precise requirements as possible. The requirements are typically gathered in a face to face meeting where Analyst(s) note down the points and later transform them into written Requirements in the form of a document. Different Analysts have different styles of documenting the requirements. The document so created is passed on further to different teams, such as Software Developers, to develop the Software. Because of different knowledge level, assumptions, abbreviations, the way of interpretation; or because of any incompleteness or inconsistency, the requirements may be modeled differently than expected. This may further lead to the development of the Software with missing or misinterpreted requirements. Suggesting ways to improve this scenario is the objective and scope of this paper. The paper will first explore ways to standardize the way the document is written. The standardization will go a long way in creating, maintaining and understanding the document. Further, it will also help us to bring some Automation where a proposed Finite State Automaton will review the document and generate errors wherever it will find the document deviating from the standard way of writing the document as proposed in this paper. We will also discuss on generation of Use Cases and Classes. All these will result in putting in less effort, resources and getting a more improved output.

Key-Words: - Requirements Engineering; Finite State Automaton; Dictionary; Use Case; Class

1 Introduction
Today’s Software Systems are more sophisticated, comprehensive, efficient; and perform more critical and complex tasks than their predecessors. Like every other project from every span of human life, developing Software also needs careful planning, design, development and testing before it gets deployed. So, when we say that the Necessity is the mother of all inventions, we need to know all such necessities for which the Software needs to be developed. These necessities are nothing but the requirements which the Software should be able to fulfill. A team of Analysts interview the customer and figure out the customer’s demands and requirements in the form of tasks and functions which the Software should be able to perform. The requirements also define the features the Software should have.

Analysts prepare a Software Requirements Specification (SRS) document based on the information they get from the customer. This SRS becomes the bible for the rest of the software development phases and the teams. The designer will design the system based on the specification in SRS. Developers will develop the code for the functionalities specified in SRS and Testers will test the software to check if it meets all the criteria as documented in SRS.

So, we see that the SRS plays an important role in the whole software development process. However, it is written by a different team and worked upon by different other teams. Everyone has their own assumptions and interpretations; and in the end the Software may behave differently than its intended purpose. For a discussion on the automatic analysis of requirements to detect such potential quality problems, please refer to [1], [2], [3] and [4].

Here, in this paper, we propose some standardization rules to give a general and yet a unique format to the SRS in order to improve its quality. These rules will preserve the current appearance of the document and at the same time will modify its structure to make it look like a program written in a High Level Programming Language. This document can then be
passed to a Finite State Automaton, which will review the document for the standard structure and the errors, if any. A detailed discussion to use the Finite Automaton can be found in [8]. We will also discuss, in brief, about the generation of Use Cases and Classes. A detailed discussion can be found in [9].

2 Document Standardization

A number of approaches in the form of best practices have been proposed by different users / communities such as [5], [6] and [7]. Some of the Standardization Rules being proposed here are:

- Creating a user defined Dictionary of Actors, Attributes and Actions. This Dictionary will have the definition for each Actor, Attribute and Action. It will also serve the purpose of a Global Library for our so called High Level program embedded in the SRS. The concept of Dictionary is inspired from [8].
- Different names should not be used for the same Actor or Attribute. For example, if Supervisor and Manager mean the same, it is better to define only one of them in the Dictionary and use throughout the SRS.
- Do not refer to an Actor, Attribute and / or Action for which no definition is given in the Dictionary. Add the definition in the Dictionary if such a case arises.
- Do not refer to an Actor, Attribute and / or Action in a way which violates the definition given in the Dictionary. Update the definition in the Dictionary if such a case arises.
- Assigning a unique identifier for each and every requirement in the SRS. This identifier will keep the Document look of the SRS and it will also provide us a separate Program like structure in the SRS.
- Each identifier will be unique for each requirement. There will be a single and separate sentence for each requirement.
- Each sentence should be complete, without any abbreviation and with as much detail as possible.
- Only Active voice sentences will be allowed for easy identification of Actors and associated Actions.
- Only Nouns should be used to eliminate the possible confusion associated with the use of Pronouns.
- Avoid use of subjective words, phrases which can be interpreted differently. For example, avoid use of “May be”.

3 Working with Dictionary

To understand the concept of Dictionary, consider the example of a Program written in a High Level language. The program will have Declarations for various Variables being used in the Program along with a Global Library of functions. When the compiler will find these Variables in the program down further, it will be aware of these Variables and for each such Variable, it will also match the Data Types. Deviations or violations, if any, will promptly be informed by the Compiler before proceeding any further.

Similarly, we propose here to build a user defined Dictionary of Actors, Attributes and Actions. All the entities related with our Software, such as Systems, Up Stream & Down Stream Systems, Processes, Business Rules, different User Roles, Data Elements, etc; can be declared and defined here. When our Finite Automaton will review the document, it will be aware of each system entity because of this Dictionary. An error message will be generated when an entity is being used in the document but is not declared in the Dictionary. A warning message can also be generated for an entity which is declared in the Dictionary but never used in the Document.

The use of the entities in the document should be in synchronization with the definitions given in the Dictionary. When any new entity needs to be used or the definition for an existing entity needs to be modified, the Dictionary should be updated for all such cases.

4 Writing the Requirements

As we have mentioned in the Document Standardization above, the Requirements will be written using the proposed best practices.

The Requirements can be classified into the following categories:

- System Requirements: This type of requirement states the actions that the proposed System must be able to do. We can assign a unique variable for each such requirement beginning with SysReq.

For example:

SysReq1: Payroll System will process the Salary on 15th of every month.
SysReq2: Library System will send a reminder mail to user before 3 days of the Due Date.

- **User Requirements**: This type of requirement states the actions that either the user must be able to do or the system should provide the ability to user to do so. We can assign a unique variable for each such requirement beginning with UseReq.

For example:

UseReq1: Library System should allow user to re-issued the book twice.
UseReq2: Admin should be able to add new users.

- **Conditional Actions**: This type of requirement states the actions which should be executed / denied by the System based on certain conditions. We can assign a unique variable for each such requirement beginning with ConAct.

For example:

ConAct1: The Banking System should deny the Debit transaction if the transaction will cause the account balance to become negative.
ConAct2: Firewall should allow a program to execute if the program is in the list of Trusted Programs.

- **Conditions on Attributes**: This type of requirement states various conditions on different attributes. We can assign a unique variable for each such requirement beginning with ConAtt.

For example:

ConAtt1: Credit Score must always be above 600.
ConAtt2: A 4 wheel SUV must have a 3500 CC V8 engine.

- **Business Policies**: This type of requirement states various Business Policies or Business Rules which the system should comply with. We can assign a unique variable for each such requirement beginning with BusPol.

For example:

BusPol1: System will calculate Provident Fund on Basic Salary.

BusPol2: The Banking System will deduct USD 60 annually for the Credit Card fee.

The requirements can be nested in If - Else statements or any other Control Structure as long as we are following the Best Practices proposed in this paper. The Finite Automaton will identify the Requirements based on the identifiers as explained above. It will not process those parts of the document which don’t have an Identifier associated with them. This way, we will be able to preserve the existing format of the Document and yet, at the same time, we will embed our requirements into the form of a High Level Language Program into the document. The Finite Automaton will then take help of the Dictionary to process these requirements.

5 Finite Automaton at Work

As we stated earlier, the Finite Automaton can provide us some amount of Automation and can review the document for us. It will first scan the document and will recognize the tokens in the form of Actors, Attributes and Actions as defined in the Dictionary. Then, it will run through the State Machine to review the document and verify for the standard and structure. It will generate messages when it will encounter an error. Let us now state its working in more details below:

5.1 Lexical Analysis

As with any other High Level Language Program, tokens will be generated first. Each sentence of the document which is preceded by the identifiers will be passed to the Lexical Analyzer which will generate the token and recognize them into Actors, Attributes and Actions token. Words such as Will, Should, Must be recognized as Supporting Phrases while the articles A, An and The will be omitted. Other letters or words such as To, From will also be omitted.

The tokenized requirements can then be passed to the Syntax Analyzer. Based on the Supporting Phrase or the type of the identifier found before the requirement, a different State machine can be used to verify the requirement for the structure, standard and completeness.

5.2 Syntax Analysis

As we said, we can have different machines for requirements having different types of identifiers or Supporting Phrases. The tokens will be fed to the State Machine. A successful run will take the Sate
Machine to an acceptable Final State whereas an improper requirement will take the State Machine to an Error State. Each Error State will generate a message to let user know what needs to be fixed.

For the explanation purpose, we can define a State Machine as a 5 tuple \((\Sigma, S, S_0, \delta, F)\) where:

- \(\Sigma\) is the input alphabet. Here the input consists of tokens recognized by Lexical Analyzer as Actor, Attribute, Action or Supporting Phrase.
- \(S\) is the set of all the states of the State Machine.
- \(S_0\) is the start state of the State Machine.
- \(\delta\) is the transition function which drives the State Machine through the inputs received.
- \(F\) is the subset of \(S\) and denotes the set of Final States.

Let’s say we are using below mentioned State Machine, as shown in Table 1, for a System Requirement.

**Table 1: Finite Automaton for sysreq**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Actor</td>
<td>Intermediate State</td>
<td></td>
</tr>
<tr>
<td>S0</td>
<td>Attribute</td>
<td>Missing Actor State</td>
<td>Error State: Generate missing Actor message</td>
</tr>
<tr>
<td>S0</td>
<td>Supporting Phrase</td>
<td>Missing Actor State</td>
<td>Error State: Generate missing Actor message</td>
</tr>
<tr>
<td>S0</td>
<td>Action</td>
<td>Missing Actor State</td>
<td>Error State: Generate missing Actor message</td>
</tr>
<tr>
<td>Intermediate State</td>
<td>Supporting Phrase</td>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>Action</td>
<td>Action State</td>
<td>Acceptable Final State</td>
</tr>
<tr>
<td>S2</td>
<td>No tokens For the input</td>
<td>Missing Action State</td>
<td>Error State: Generate missing Action message</td>
</tr>
</tbody>
</table>

To understand the working of the Finite Automaton, let’s take some examples.

**Example 1:**

SysReq1: Payroll System will process Salary.

The Lexical Analysis will identify this as a System Requirement. Let’s also assume that the Payroll System and Process Salary are there and defined in the Dictionary as Actor and Action respectively.

The Lexical Analysis will recognize Payroll System, Will and Process Salary as tokens.

The State Machine will traverse through S0, Intermediate State, S2 and Action State.

**Example 2:**

SysReq1: Payroll System will process Salary.

The Lexical Analysis will identify this as a System Requirement. Let’s also assume that Process Salary is there and defined in the Dictionary as Action.

The Lexical Analysis will recognize Will and Process Salary as tokens whereas it will not be able to recognize Payroll System. It will generate an error prompting user to add Payroll System in the Dictionary first.

**Example 3:**

SysReq1: Payroll System will process.

The Lexical Analysis will identify this as a System Requirement. Let’s also assume that the Payroll System and Process Salary are there and defined in the Dictionary as Actor and Action respectively.

The Lexical Analysis will recognize Payroll System and Will. For Process, however, it will generate an error message saying that it is expecting Salary after the word Process.

We are not providing other Finite Automatons for other Identifiers here to keep the discussion brief.

### 6 How the Use Cases and Classes will be Generated

After the successful run of the Automaton, the document will be scanned for all the requirements. The Action part associated with an Actor can be proposed as a Use Case. For each such Use Case, a scan in the document will provide all the details and the functionalities associated with the Use Case.
All such Use Cases along with their functionalities and details can be used to identify the Classes and their Attributes. Here, we can take help of some Natural Language Processing (NLP) tools to do that. For a detailed discussion on the use of NLP for automating Requirements Engineering, please refer to [9].

We can take help of below pointers to identify the classes.

- If a Noun doesn’t have any property, it is not a Class.
- Based on the Action Phrase, In a Noun – Noun structure, if the first Noun is a Class then second Noun can be an Attribute.
- Using the simple heuristic searches, some more attributes can be found such as height of, temperature of etc.

Association, Aggregation and Generalization relationship among Classes can also be drawn.

For example:

- In the sentence, User has a Computer, Has is the Association between User and the Computer Class.
- Sentence such as Report Master contains Reports is showing Aggregation relationship between Report Master and Report Class.
- Sentence such as A User can have Admin user Role and Guest user Role, proposes a Base Class User Role with an inheritance for Admin User Role and Guest User Role.

7 Conclusion and Future Work

Here, we proposed how to achieve some degree of Automation using some of the best practices of the document writing. The suggested approach can be further refined. Instead of using restricted natural language, NLP tools can be used to write the requirements completely in a natural language. Finite Automaton based approach can also be extended to a Compiler approach where the requirements can be checked for logical errors. Opportunities can also be explored to be able to draw the UML diagrams.

References: