EMPIRICAL TEST DESIGN STRATEGIES USING NATURAL LANGUAGE PROCESSING

A PROJECT REPORT

Submitted By

ABISHEK T.S. 312216104001

ADITHYA VISWANATHAN 312216104002

AKASH KUMAR PUJARI 312216104007

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM 603110

ANNA UNIVERSITY :: CHENNAI - 600025

March 2020

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report titled **"EMPIRICAL TEST DESIGN STRATEGIES USING NATURAL LANGUAGE PROCESSING"** is the *bonafide* work of "**ABISHEK T.S. (312216104001)**, **ADITHYA VISWANATHAN (312216104002)**, and **AKASH KUMAR PUJARI (312216104007)**" who carried out the project work under my supervision.

DR. CHITRA BABU
HEAD OF THE DEPARTMENT
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

DR. FELIX ENIGO
ASSOCIATE PROFESSOR
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving us the strength and knowledge to do this project.

We would like to thank and deep sense of gratitude to our guide **DR. V.S. FELIX ENIGO**, Associate Professor, Department of Computer Science and Engineering, for her valuable advice and suggestions as well as his continued guidance, patience and support that helped us to shape and refine our work.

My sincere thanks to our project mentor **Dr. KAUSHIK RAGHAVAN**, Senior Project Officer of the Department of Computer Science and Engineering IIT Madras, for his words of advice, encouragement and valuable suggestions throughout the project

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also express our appreciation to our **Dr. S. SALIVAHANAN**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of our project work. Finally, I would like to thank our parents and friends for their patience, cooperation and moral support throughout our life.

Abishek T.S. **Adithya Viswanathan** **Akash Kumar Pujari**

ABSTRACT

With the rise in the role played by computers and their ubiquity, both developers and consumers share responsibility for improvements in the digital platform. Thus, there is a need to reduce the load on both parties with the help of automation. This proposed system, provides a simplistic interface where users can submit issues with applications whereas developers are provided with test cases before writing off on the solutions to these issues. It aims to intelligently identify pre-existing similar issues if any to reduce redundancy, identify system requirements unsatisfied by these issues as well as identifying the criticality of the issues. It also seeks to generate test cases for functional requirements. The system uses a DAN model developed by google for sentence similarity to detect similar issues, providing an accuracy of upwards of 90% for an in-house prepared test set of more than a 100 entries. It also uses sentence dependencies and a SMT solver, for identifying POS as well as the subject, object, verb, variable and its corresponding parameter in a functional requirement and generating appropriate test cases. The system represents a definite move forward towards complete autonomy of the aforementioned process.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

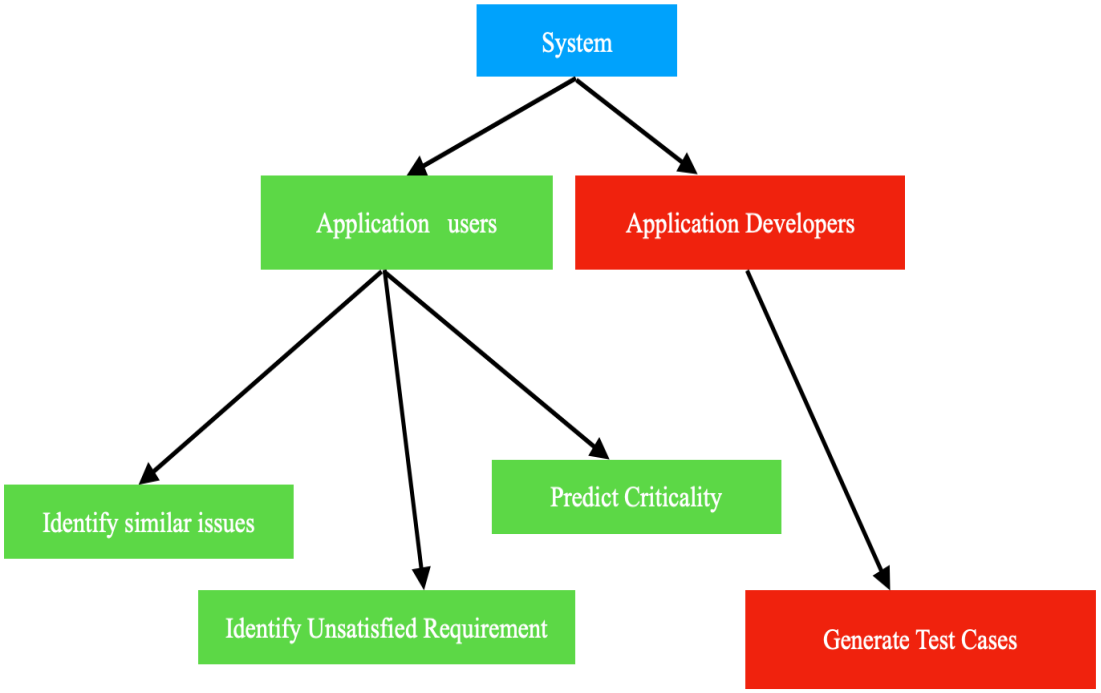
INTRODUCTION

It is common knowledge that software development is an iterative process, with the developers constantly adding functionalities, fixing bugs, analysing usage and holistically improving the product with each iteration. Almost every digital service provider such as Apple, Mozilla Foundation, Google, Amazon etc. all diagnose issues submitted by their users and products aimed at refinement.Thus, one can understand that a combined effort from both users as well as developers is essential. The problem however doesn't end there. Every time a new version of a software/application is released, it is imperative that it is subject to rigorous application testing, which is aimed at finding bugs in the software as well as in its dependencies.

The testing phase also includes creation of a test plan, the process of Test Case Estimation as well as Test Case Generation explained in detail in the work done by Myers Glenford J[14] On an average, the development of a test case takes about 10 minutes, and in general, development of a test plan without test cases and its review can take about 2-3 days. A cumulation of these factors create a necessity for a system, that if not removes, at least reduces the time and energy spent for this.

Our whole project can be divided into two abstract working components, the first seeking to help users by providing recommendations such as pre-existing similar issues, identifying system requirements unsatisfied by these issues and predicting the criticality of the issue. This is done by using sentence similarity models which

FIGURE 1.1: Abstract System Architecture



identify common semantics between sentences to match them. The second, seeking to help developers by generating a set of test cases based on the functional requirements when the corrections of the issues raised, need to be verified.

In the next chapter, we get a technical understanding of the workings of the sentence similarity model. In chapter 3 we will look into the NLU, a subset of NLP which is used for predicting the criticality. In chapter 4 we look at sentence dependencies, which play a major in pre processing and post processing of the input and output respectively to the SMT solver which is discussed in chapter 5.

CHAPTER 2

LITERATURE REVIEW

This literature survey aims to provide some valuable insight into 4 major topics that play an important role in the project, namely Sentence Similarity, Natural Language Understanding, Sentence Dependency and SMT.

The last 7 decades have seen a rapid increase in the importance of Natural Language Processing and its various applications. The idea of Natural Language Processing within the realms of Artificial intelligence emerged from the concept of the imitation game, introduced by Alan Turing in the year 1950. From then on, NLP has evolved into the enormous field that it is today. Sentence Similarity is an important subset of NLP which itself has several applications today. It evolved from the computation of extended cosine distance between the vectorised forms of sentences [6]

to the use of various Neural Networks to group similar sentences. The current trend is the use of LSTM, to measure similarity between sentences [7]

However, there are a few disadvantages to the use of such Recurrent Neural Networks. They need a large amount of training time. The non-linearities and matrix products at each node of the parse tree are expensive, especially when the dimensionality increases. They also require consistent syntax between training and test data due to their reliance on parse trees and thus cannot effectively incorporate out of domain data.

There is also a steady increase in the use of NLU, which is a subset of NLP to create entities such as chatbots, to provide tailored responses to inputs. A theoretical

interpretation of Natural Language Understanding can be obtained from the work done by Roger C. Shank [11] as early as 1972. Rasa [8], which provides an open source conversational AI framework for building contextual assistants is one of the industry's leading service providers. Their core functionalities can be ported for obtaining responses based on key words, by training with developer provided data corpus.

Every word in a sentence is grammatically linked to other words in the sentence. This phenomenon is known as sentence dependency and is very useful in extracting valuable information both semantically and structurally. In linguistics, a treebank is a parsed text corpus that annotates syntactic or semantic sentence structure. The construction of parsed corpora in the early 1990s revolutionised computational linguistics. The exploitation of tree bank data has been important ever since the first large-scale tree bank, The Penn tree bank [9]. Dependency parsing is strongly linked to POS tagging, which is also a complicated task, because the contexts of the same words vary with sentences. An overview of increasing the accuracy of tagging words is given in the work done by Christopher D. Manning [10].

Satisfiability Modulo Theories is one of the core branches of theoretical computation. The motivation to develop these arose due to the limitations of SAT solvers which only dealt with boolean levels and the translation of a problem to boolean logic can be expensive. The primary goal of research in SMT, is to create verification engines that can reason natively at a higher level of abstraction while still retaining the speed and automation of today's boolean engines. The language of SMT solvers is First-order Logic(FOL) where the symbols are divided into logical symbols and non-logical symbols or parameters. In the scope of this

research, SMT solvers are used to provide a model that satisfies a set of asserted constraints.

The work done by Dwarakanath Anurag and Shubhashis Sengupta [13] provides a comprehensive method for the generation of test cases using link grammar. However the collusion of sentence dependencies and SMT solver will serve as a powerful tool to generate test cases for scenarios with multiple dependant constraints.

CHAPTER 3

SENTENCE SIMILARITY

Sentence Similarity or Semantic Textual Similarity is a measure of how similar two pieces of text are, or to what degree they express the same meaning. This is broadly useful in obtaining good coverage over the numerous ways that a thought can be expressed using language without needing to manually enumerate them. It is a widely used in the domain of Natural Language processing. The process of identifying similar sentences usually entails that the text is first converted into high dimensional vectors. This process is known as embedding.

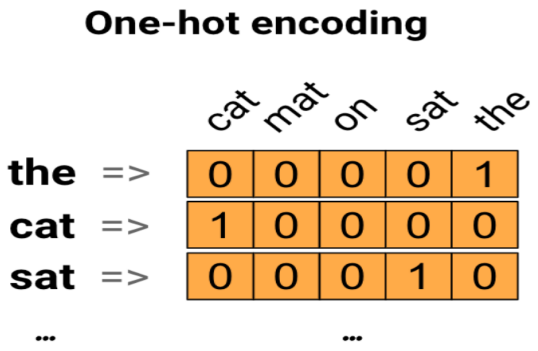
3.1 Embedding of Text

There are 3 strategies that one can use to embed text

- One-hot encoding - Here, to represent each word, we will create a zero vector with length equal to the vocabulary, then place a one in the index that corresponds to the word. This process is very inefficient. Consider the scenario where we have 10,000 words in our vocabulary. If we need to encode one word, then we need to create a vector where 99.99% of the inputs are zero.

- Encode each word with a unique number - In this approach, we try to encode each word using a unique number. Thus, instead of a sparse vector, we now have a dense one. However, this is not recommended as well because it does

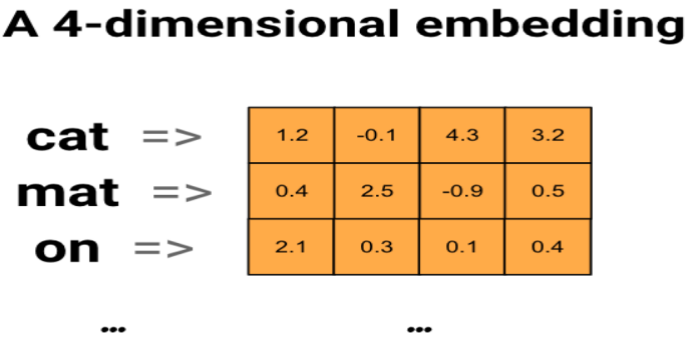
FIGURE 3.1: One Hot Encoding



not properly represent the relationships between the words and is also very difficult to interpret.

- Word Embeddings - This gives us a way to use an efficient, dense representation in which similar words have a similar encoding. The most important factor to be considered here is that we do not have to manually specify the encodings by hand. These can be trainable parameters (weights learned by the model during training).

FIGURE 3.2: Word Embedding



The model used here is the universal-sentence-encoder model developed by Google [4]. The model is trained and optimised for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide

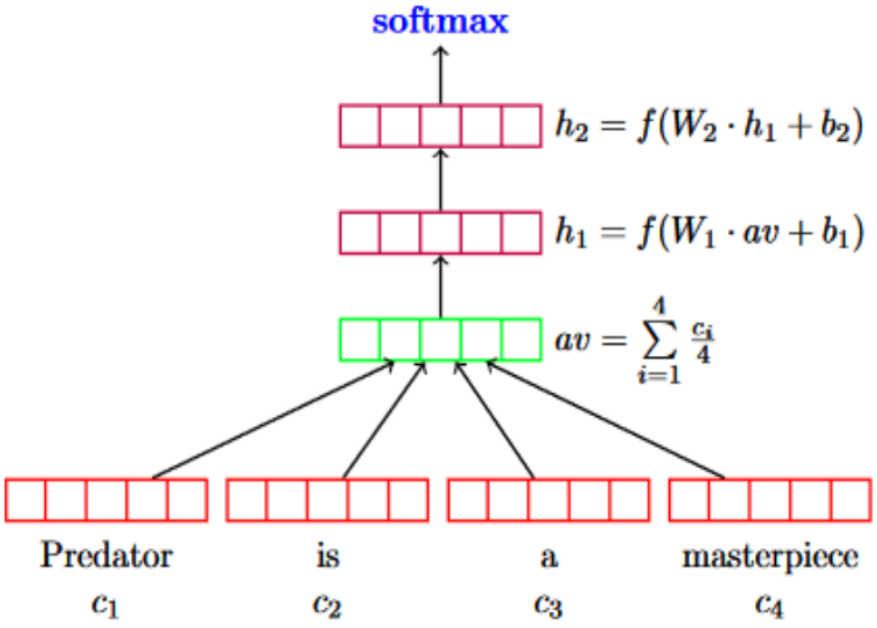
variety of natural language understanding tasks. The input is variable length English text and the output is a 512 dimensional vector. The universal-sentence-encoder model is trained with a deep averaging network (DAN) encoder.

3.2 Understanding Deep Averaging Networks

A basic idea of DAN can be obtained from the work done by Mohit Iyyer et al [3]. The intuition behind feed forward neural networks is that, each layer learns a more abstract representation of the input than the previous one. Thus with respect to textual inputs, we expect each progressive layer of the model to magnify small but meaningful differences in the word embeddings average. To provide a concrete example for the same, consider the sentence **"I loved Michael Caine's performance in the movie interstellar"** *s1*. Replacing the word **"loved"** with the word **"liked"**, we generate another sentence *s2* and replacing the word **"loved"** with **"despised"** we generate *s3*. The vector averages of these sentences are almost identical. However the averages associated with the synonymous sentences s1 and s2 are slightly more similar to each other, than to s3's average.

Improving the robustness of the Deep Averaging Network can achieved by performing word dropouts. Dropout regularises the neural network by randomly setting hidden and/or inputs to zero with some probability p [1, 2] and reduces overfitting. Given a neural network with n units, dropout prevents overfitting by creating an ensemble of 2^n different networks that share parameters, where each network consists of some combination of dropped and undropped units. Instead of dropping units, a natural extension for the DAN model is to randomly drop

FIGURE 3.3: Sample DAN Structure



word tokens' entire word embeddings from vector average. Using this method, the network can be made to see 2^x token inputs for each input x.

The word embedding model used was trained with unsupervised data sourced from a variety of web sources such as Wikipedia, web news, web question answer pages and discussion forums. This unsupervised learning was then augmented with supervised data from the Stanford Natural Language inference corpus.

Practical use of this module requires a download of a 1GB file to be downloaded the first time it is instantiated. Thus the time taken to load may vary with the user's network speed. However because the modules are cached by default, reloads do not consume as much time. Furthermore, once the module is loaded onto memory, inference time should be relatively fast.

In general Text preprocessing involves steps such as tokenisation, removal of HTML tags, removal of extra white spaces, conversion of accented characters to ASCII characters, expansion of contractions, removal of special characters, normalisation (converting text to lower case or upper case), convert number words to numeric form, removal of numbers, removal of stop words and Lemmatisation. However, this module performs best effort text input pre-processing. Thus it is not required to pre-process the data before applying it to the module.

3.3 Recommendation by the System

When a user uses our system, they try to submit issues/bugs which they have identified while using a particular software. There is a chance that another user may have already submitted an issue regarding this problem. Thus, the system must use sentence similarity models to identify these similar issues and prompt the user asking them if they still want to submit. The system also uses sentence similarity to identify requirements that have not been satisfied by these issues. The system based on DJANGO, uses AJAX calls to dynamically prompt helpful information to the user.

From the example given in Figure 3.4, we can see that when an issue **Page Load times are slow**, we the system matches it with an already existing issue, **Speed up page load/reload times** even though there is considerable difference in the way both the sentences are structured.

There also exists an option for the users to register if the predictions by the system were right, which can then be stored and used later of analytical purposes.

FIGURE 3.4: Sample Issue Recommendation

CHAPTER 4

RASA and NLU

4.1 Natural Language Processing

Natural Language Processing, is a subfield of linguistics, computer science, information engineering and artificial intelligence concerned with the interactions between computers and human(natural) languages, in particular, how to program computers to process and analyse large amounts of natural language data. Natural Language understanding is a subtopic of natural language processing, that deals with machine reading comprehension. As of the time of writing this thesis, March 2020, there is considerable interest in the field because of its application to automated reasoning, machine translation, question answering, news gathering etc. To understand why NLU was used to create this part of the system, it is imperative that one understands the necessity that the system was built upon. When a user inputs an issue, the system had to intelligently categorise the issue into levels of criticality. This required the extraction of specific keywords and taking a course of action according to these.

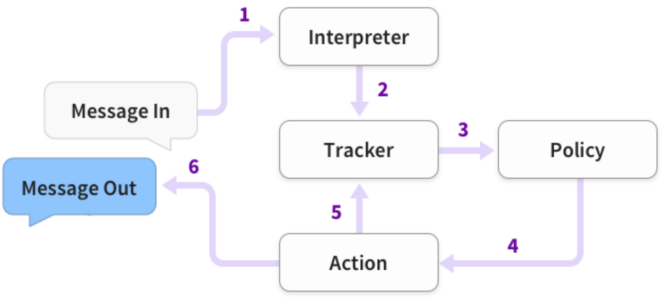
4.2 RASA

Incidentally, chatbots performed this very same task, just for a different use case and thus their functionality was ported into our project to serve a different purpose. RASA provides a open source machine learning framework to help

build contextual AI assistants and chatbots and also enable scaling with their enterprise grade platform.

The steps involved in how a response to a message is obtained are:

FIGURE 4.1: RASA Architecture



1. The message is received and passed to an Interpreter, which converts it into a python dictionary including the original text, the intent, and any entities that were found. This part is handled by NLU.

2. The Tracker is the object which keeps track of conversation state. It receives the info that a new message has come in.

3. The policy receives the current state of the tracker

4. The policy chooses which action to take next.

5. The chosen action is logged by the tracker.

6. A response is sent to the user.

The basic work performed by rasa is known as intent classification, which takes a sentence such as, ”I am looking for a Mexican restaurant in the centre of town” and returns structured data such as,

```
  {
  "intent": "search_restaurant",
  "entities": {
    "cuisine" : "Mexican",
    "location" : "center"
  }
}
```

Rasa stories are a form of training data used to train Rasa's dialogue management module. A story is a representation of a conversation between a user and an AI assistant, converted into a specific format where user inputs are expresses as corresponding intents while the responses of an assistant are expressed as corresponding action names. The training for Rasa is by default set to 100 epochs.

Actions are the things that the bot runs in response to the input. In terms of the system itself, Rasa is primarily designed to be run on a command line interface, however because the system is developed using python and the rasa actions must run in the background, only to return the predicted criticality, the parent python program sends the message/issue inputted by the user as a HTML localhost request with the message in a JSON format. Web hooks are used to obtain the object returned by Rasa, which is also in JSON format. Web hooks provide a mechanism where by a server-side application can notify a client side application when a new event (that the client-side application might be interested in) has occurred on the server.

As an example, consider a project which has 3 levels of criticality for its issues, moderate, high and critical respectively. By associating key words such as

‘improve’, ’slow’ and ‘crash’ to the 3 categories respectively in the NLU markdown file, we associate specific words with intents. Consider the same example given in Chapter 3 Section 3, represented by Figure 4.2. Given that the issue being input has the word slow, the predicted criticality is ’high’.

FIGURE 4.2: Sample Criticality Recommendation

CHAPTER 5

DEPENDENCY PARSING

Dependency parsing is the task of extracting a dependency parse of the sentence that represents its grammatical structure and defines the relationship between "head" words and words which modify those heads. A Dependency parser transforms a sentence into a dependency tree. A dependency tree is a structure that can be defined as a directed graph with 'V' nodes corresponding to the words and 'A' arcs corresponding to the syntactic dependencies between them. Labels are also attributed to dependencies called relations. These relations give details about the dependency type.

A very important part of NLP that must be addressed before moving on to the

FIGURE 5.1: Types of Dependencies

	Nominals	Clauses	Modifier words	Function Words
Core arguments	nsubj obj iobj	csubj ccomp xcomp		
Non-core dependents	obl vocative expl dislocated	advcl	advmod* discourse	aux cop mark
Nominal dependents	nmod appos nummod	acl	amod	det clf case
Coordination	**MWE**	**Loose**	**Special**	**Other**
conj cc	fixed flat compound	list parataxis	orphan goeswith reparandum	punct root dep

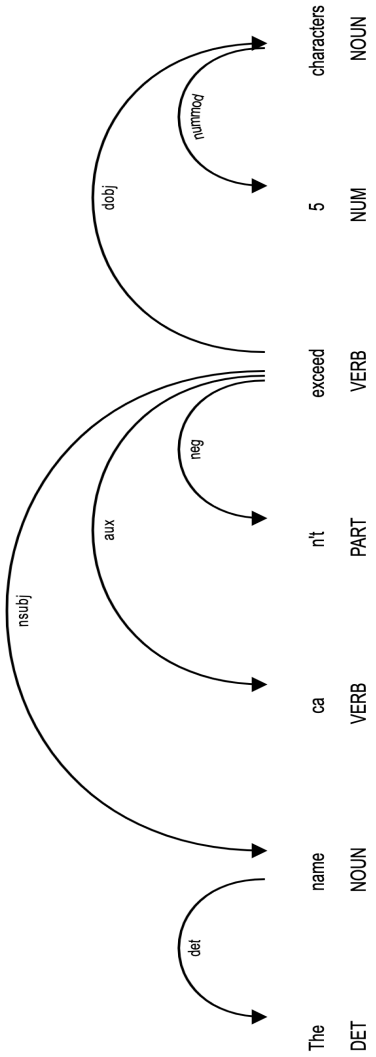
working of sentence dependencies is POS Tagging. This refers to Parts Of Speech Tagging and it is the process of marking up a word in a corpus to a corresponding port of a speech tag, based on its context and definition. This task is not straightforward, as a particular word may have different part of speech based on the context in which the word is used. For example, in the sentence "Give me your answer", answer is a Noun, but in the sentence "Answer the question", answer is a verb. In the case of the spaCy python module, the fastest syntactic parser in the world [12], which is used in the system, a statistical model comes in, which enables spaCy to make a prediction of which tag or label most likely applies in this context. A model consists of binary data and is produced by showing a system enough samples for it to make predictions that generalise across the language.

FIGURE 5.2: Universal POS Tags

Open class words	Closed class words	Other
ADJ	ADP	PUNCT
ADV	AUX	SYM
INTJ	CCONJ	X
NOUN	DET	
PROPN	NUM	
VERB	PART	
	PRON	
	SCONJ	

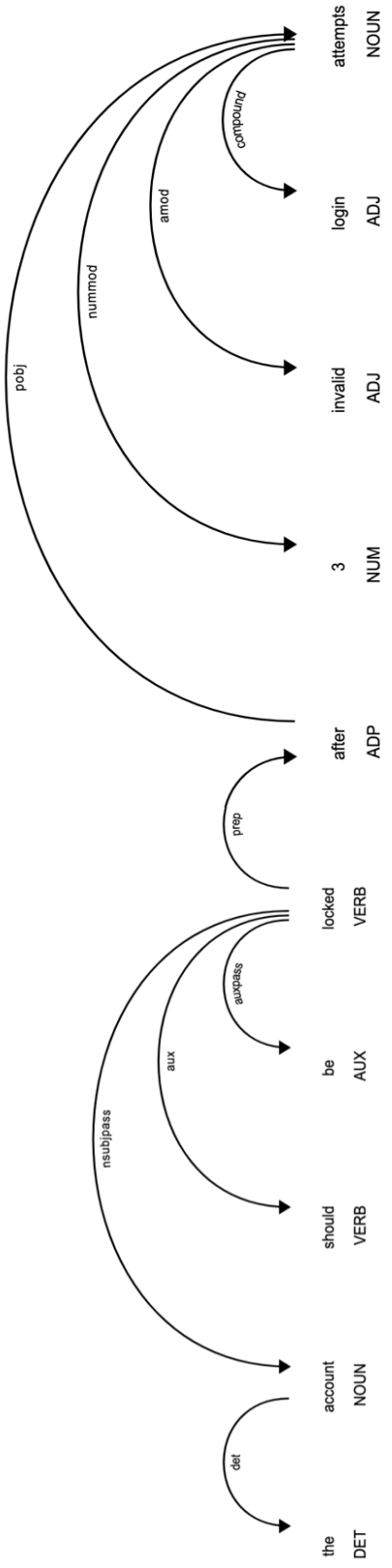
The scope of the system is limited to the generation of test cases for functional requirements which have numerical parameters to be satisfied. With this in mind, it is easy to understand that the most important dependency is the nummod dependency which stands for numeric modifier. Let us first define a few terms which will help us better understand how information is extracted from a sentence using dependencies.

FIGURE 5.3: Sentence Dependency



- Variable - An entity that is constrained by the parameter
- Parameter - Numerical value which servers as the start of the search
- Subject - The subject of the sentence
- Object - The object of the sentence
- Verb/Auxiliary term - The action word that is linked to both the subject and the object

FIGURE 5.4: Example Dependency



We identify the number in the sentence and set it as the parameter while the head term it is the nummod of along with those linked to it, delimited by the verb are together considered to be a single entity known as the variable.

In all sentences, the action word that is linked to the subject is considered as the verb and the sentence may or may not have an object. This can be clearly understood with an example:

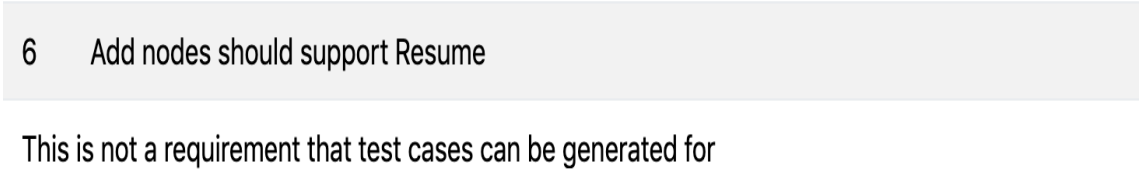
Consider the sentence **"The account should be locked after 3 invalid login attempts"** represented by Figure 5.4 Here the verb is locked, and it is linked to the word account with the dependency 'nsubpass' and thus the word account is considered to be the subject. 3 is the numerical modifier of the word attempts which in turn is linked to the words invalid and login, thus 3 is taken as the parameter while the words 'invalid login attempts' are taken as the variable.
In this example, the sentence has no object. Thus we can generalise that any word linked with a dependency containing sub-strings 'sub' and 'obj' to the verb are the subject and object respectively.

We explicitly extract this information from the sentence because once this information is passed to a SMT solver and the output is received, a legible test case must be constructed to be submitted to the developers.

Given that the probability of representing numbers as words while writing the requirements is very high, the system has the capability of identifying these and converting them to numeric form, before feeding it to the SMT solver, discussed in Chapter 6. For example, when parsing the sentence:
The battery percentage must be greater than thirty three.
the system will recognise thirty three to be the number 33. This in turn works with the python module **word2number** used along in the SMT solver.

It is also important to note that there may be requirements for which the system is not capable of generating test-cases. Thus, the system identifies these cases and notifies the user of the same.

FIGURE 5.5: Unable to Generate Test-Case



CHAPTER 6

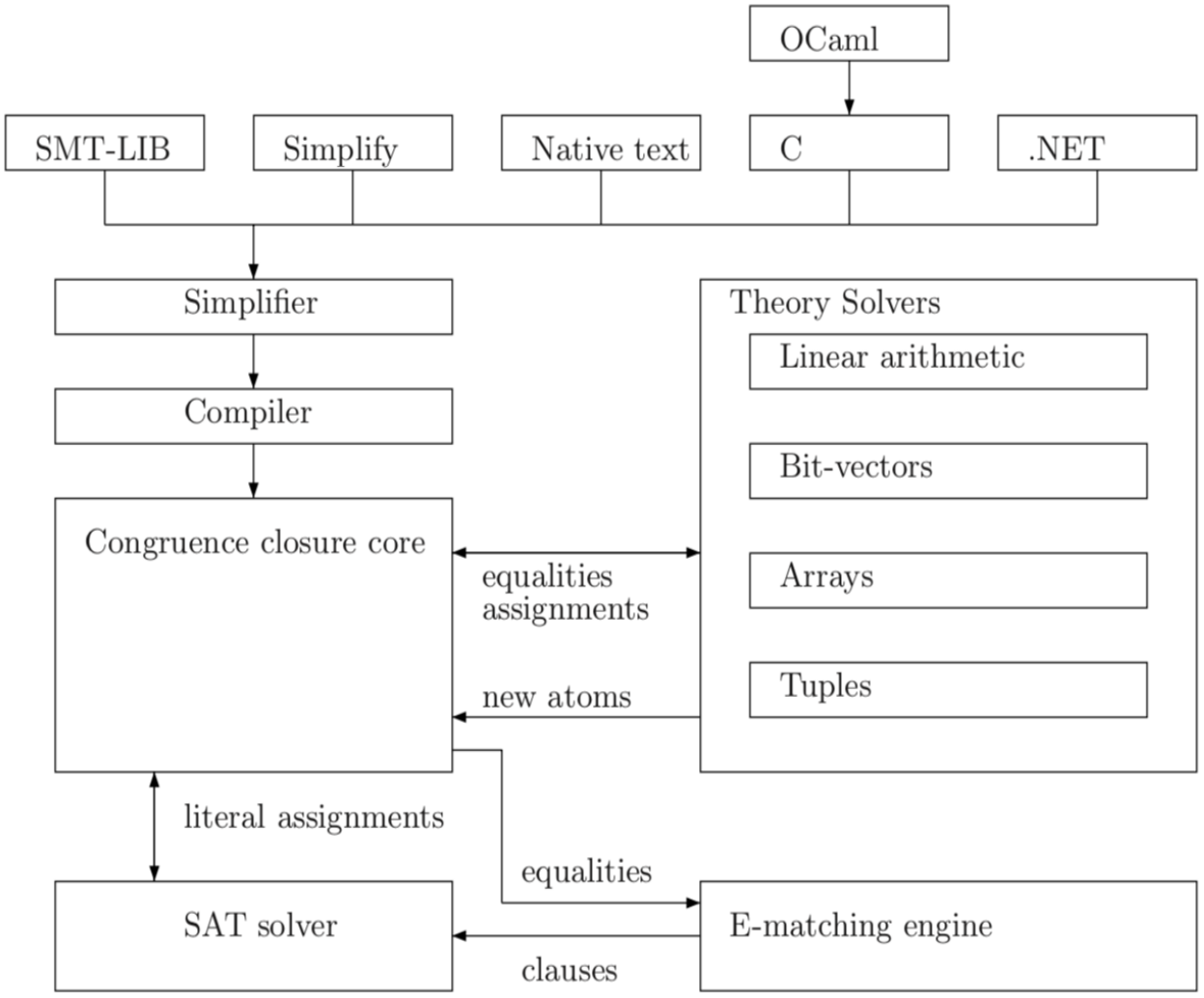
SMT SOLVER

6.1 Z3

In computer science and mathematical logic, the Satisfiability Modulo Theories problem is a decision problem for logical formulas with respect to combinations and background theories expressed in classical first-order logic with equality. There are many available SMT solvers and the system uses the Z3 [5], which has built in theories such as empty theory, linear arithmetic, non-linear arithmetic, bit vectors, arrays, datatypes, quantifiers, strings etc. It also has APIs in C/C++, .NET, Python, Java etc. It is developed by Microsoft research and is used in many applications such as software/hardware verification and testing, constraint solving, analysis of hybrid systems, security and geometrical problems. The system architecture of Z3 is as given in Figure 6.1

It terms of the use cases that the system will be subjected to, a SMT solver is required to provide a solution for the set of asserted constraints. We say that the solution is a model for the set of asserted constraints. A model is an interpretation that makes each asserted constraint true.

FIGURE 6.1: Z3 System Architecture



6.2 Test Case Generation

Given that test cases must be generated for all possible scenarios such as correct inputs, corner cases etc. even though the constraint is extracted from the input sentence using sentence dependency, the test cases are generated for all permutations and combinations. The output of the SMT solver is an assignment. This must then be used to construct a test-case that can easily be understood by the developer.

One might then wonder why go to the trouble of extracting of the constraints and various other information using dependency parsing. When the test-cases are presented in the interface, they are divided into two sets. One containing the positive and corner test cases while the other containing the negative ones.

However, if one understands the workings of Z3, they will be able to see that for a given set of constraints, the model created will always be the same. This drawback was overcome by using random selection of multiple generated test-cases.
Rather than adding constraints to the z3 solver with multiple iterations, where each constraint imposed a new rule where the model could not generate the same solution as the previous iteration, the solver was set to create a model for a fixed number of iterations, where each iteration had different constraints all together. From the final set of models created by this process, the required number of positive, negative and corner cases are chosen.

With the same example considered in Chapter 5, which is **The account should be locked after 3 invalid logins**, we see that in Figure 6.2 and Figure 6.3, which represent consecutive test-case generation requests by the developer, the test-cases generated are not the same.

FIGURE 6.2: Generated Test-Case 1

Test cases generated:

Check with:

invalid login attempts = 2

invalid login attempts = 1

---

invalid login attempts = 11

invalid login attempts = 4

FIGURE 6.3: Generated Test-Case 2

Test cases generated:

Check with:

invalid login attempts = 3

invalid login attempts = 1

---

invalid login attempts = 10

invalid login attempts = 7

CHAPTER 7

SYSTEM DESIGN, ARCHITECTURE and IMPLEMENTATION

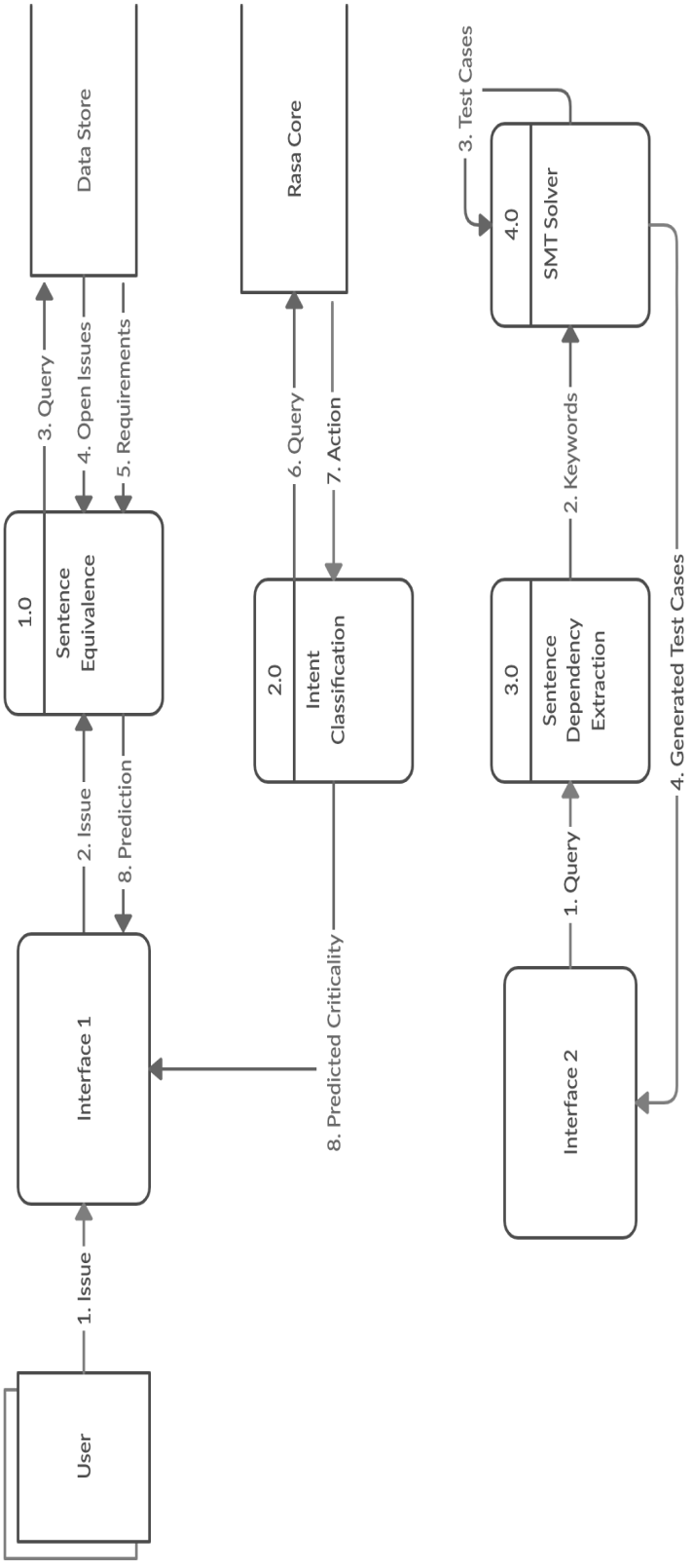
7.1 Architecture

The whole project was developed with a python base. It can be viewed as two core modules, which interact with one another using AJAX calls, namely:

- User Interface - Developed using python Django which is a high level framework that encourages rapid development and clean, pragmatic design. This HTML/CSS based interface was constructed with simplicity in mind, with its fair share of aesthetics.
- Principal program - Is responsible for initialising various dependent servers, and interacting with other subsidiary modules which are responsible for the functionalities such as identifying similar issues, predicting criticality, etc.

A comprehensive understanding of the system can be obtained by analysing the Figure 7.1. We can see that the system has two separate interfaces to cater to the needs of its different user base (i.e. application users and application developers).

FIGURE 7.1: Data Flow Diagram



7.2 Implementation

This section covers the technical aspects of the proposed system, in terms of pre-requisites and the various python modules used along with their purpose as well as the technical flow of the program.

The program runs on *python3*. The use of python2 was differed because, as of January 2020, it reached it EOL (End Of Life) status and no would receive no further official support.
To run the code, identify the directory which contains the program *manage.py* and run it with the command

```
python3 manage.py runserver
```

This in turn runs multiple subsidiary programs, and also sets up servers running on ports 8000, 5055 and 5002. Considering the directory which contains the *manage.py* python program to be the parent directory, the flow of the program can be understood from Figure 7.2 and Figure 7.3.

FIGURE 7.2: Manager Program

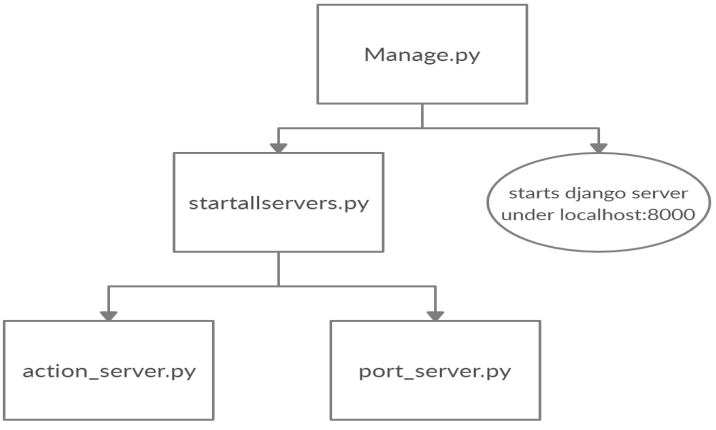
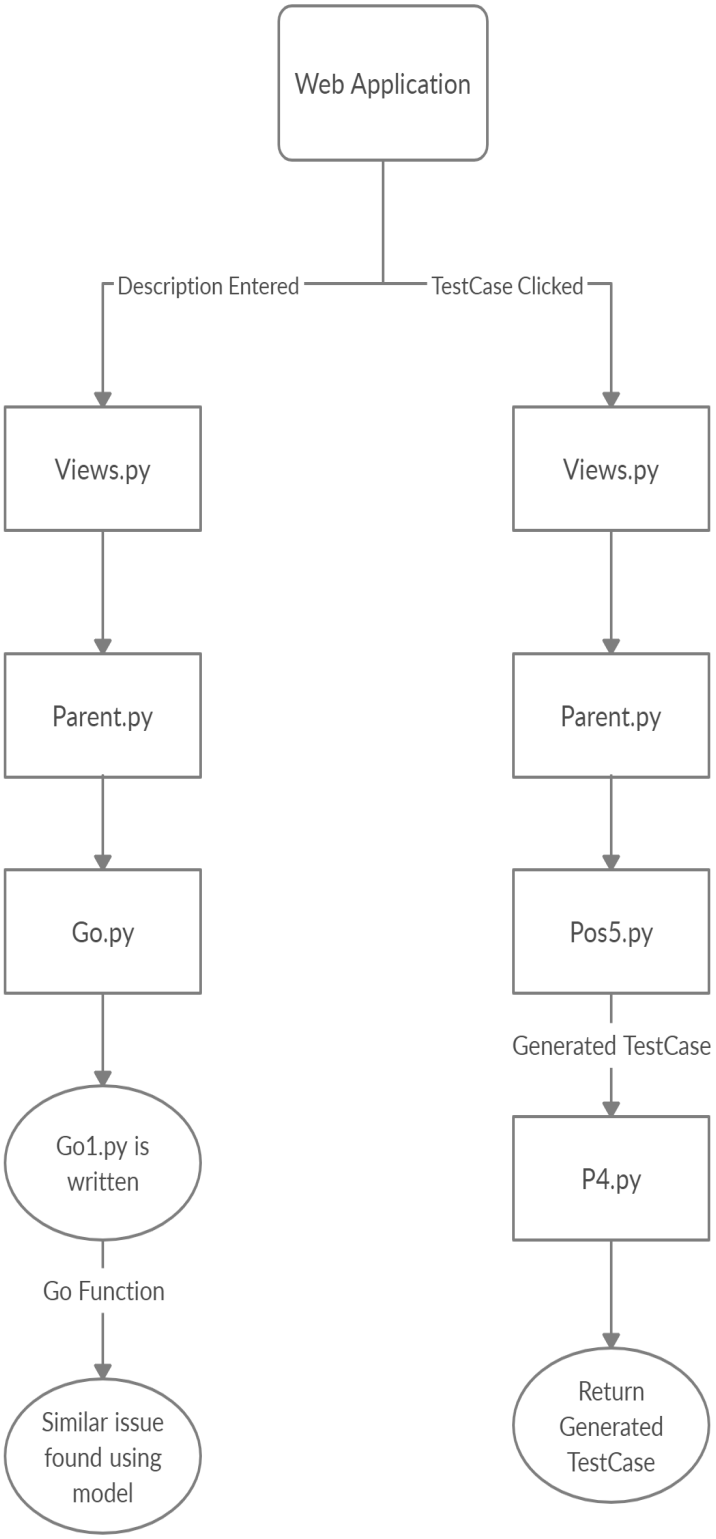


FIGURE 7.3: Program Tree



The various python packages used here are:

- os: This is used for miscellaneous Operating system calls from within python programs.
- subprocess: For executing multiple calls in parallel, such as starting up 2 servers simultaneously
- rasa: for implementing the various rasa functionalities This is set up in the system with the help of the command

  ```
  pip3 install rasa
  rasa init
  ```

  If there are any changes to the levels of criticality, addition of new keywords etc. in the project, the rasa model can be trained again using the command

  ```
  rasa train
  ```

- tensorflow and tensorflow_hub: This is used for importing tensorflow models, in the case of sentence similarity and for the creation of sessions etc.
- numpy: For various operations involving vectors, especially during word embeddings etc.
- spacy: For performing dependency parsing
- z3: For implementing the various functionalities of the SMT solver.

7.3 Working Screenshots

FIGURE 7.4: Form Submission UI

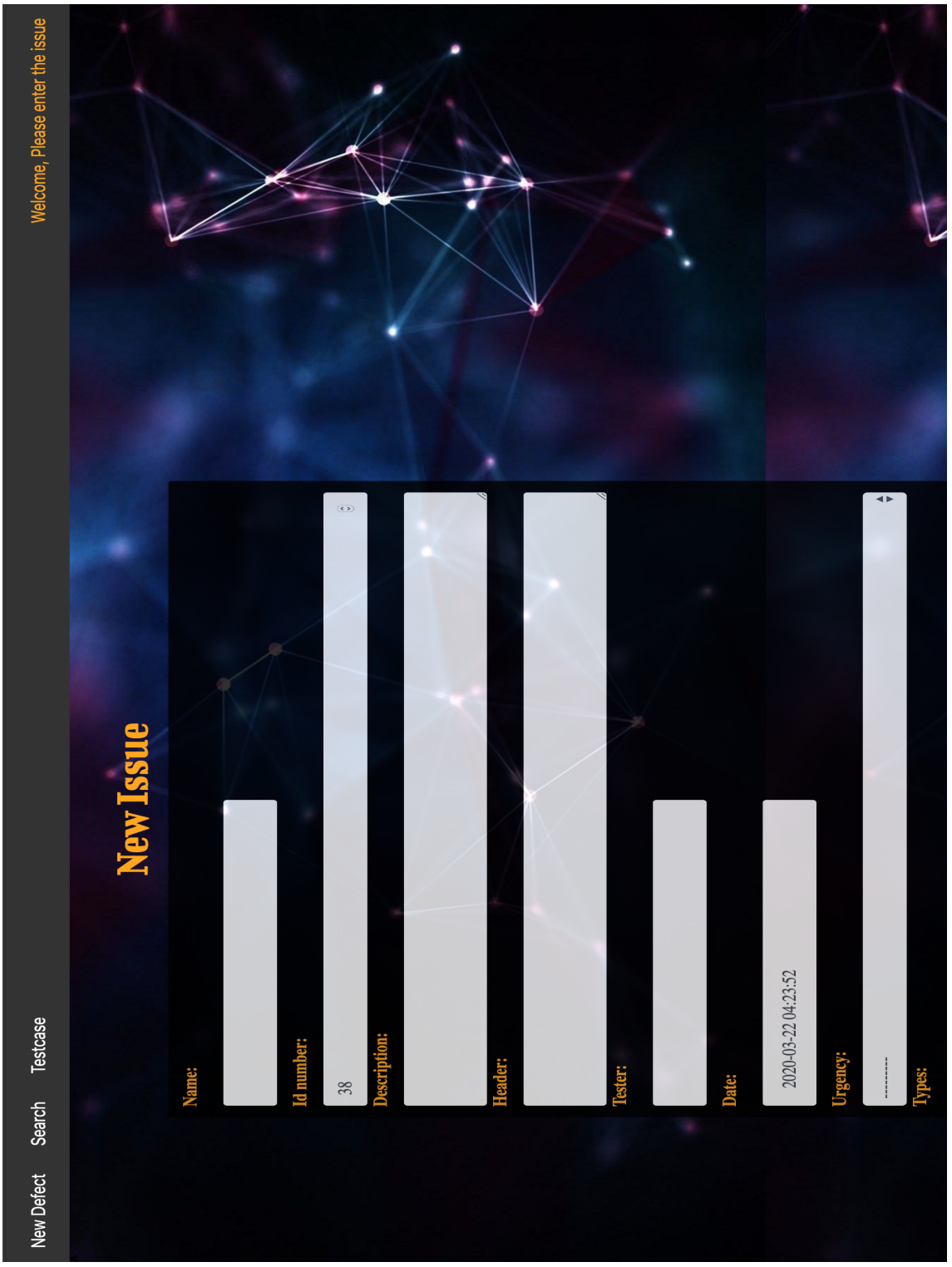


FIGURE 7.5: Form with Recommendation

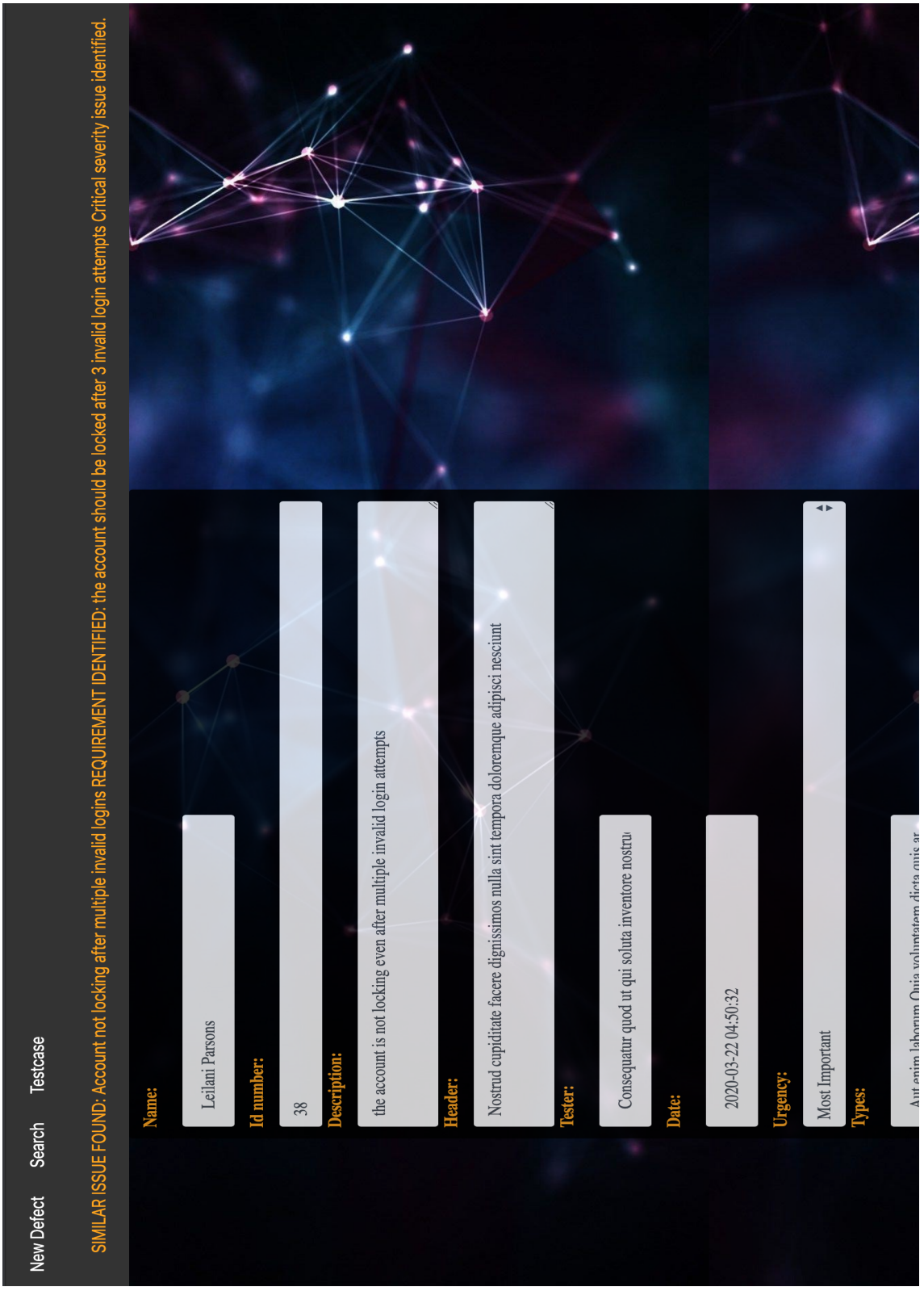


FIGURE 7.6: Search Tab for submitted Issues

FIGURE 7.7: Requirements Page

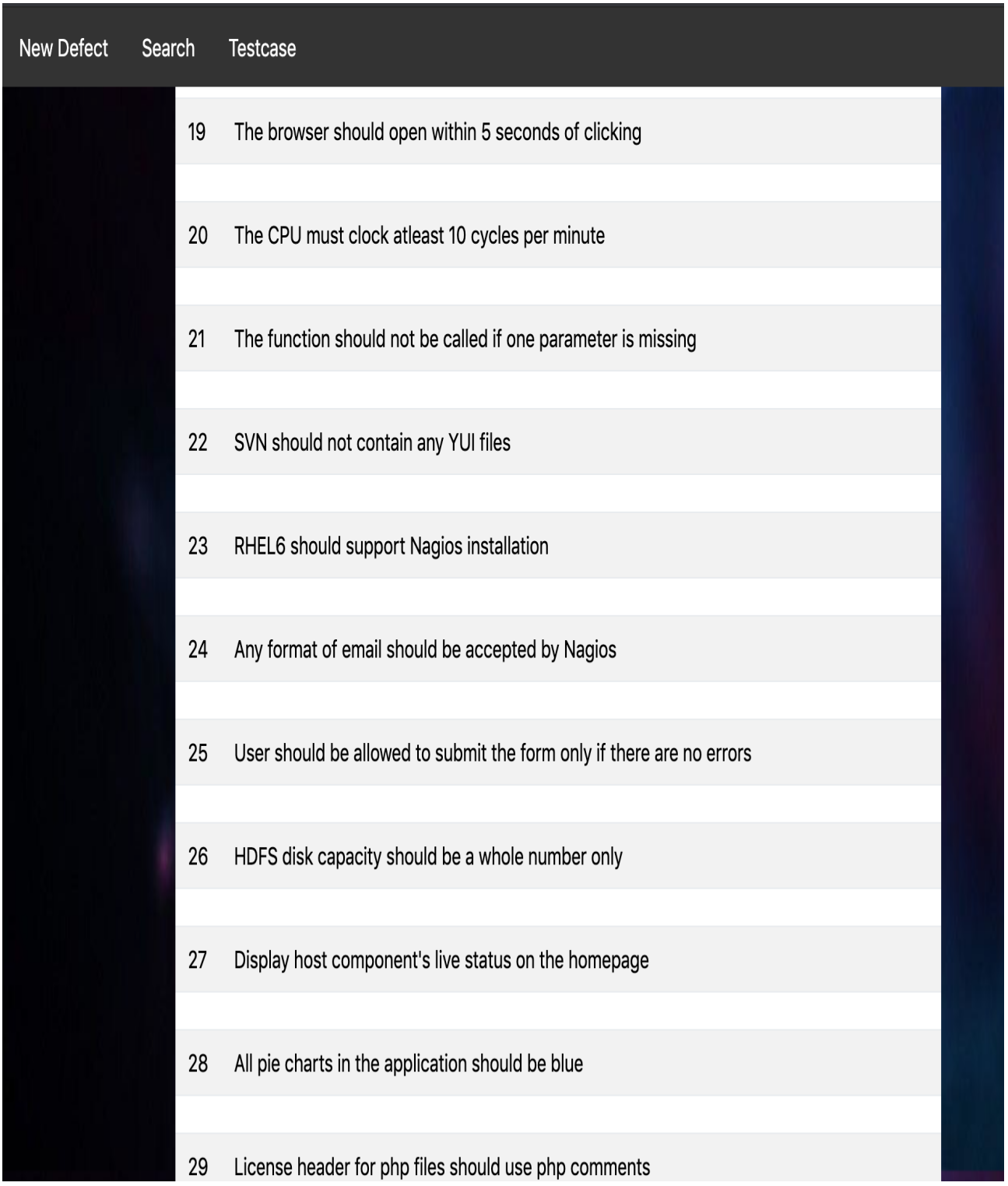


FIGURE 7.8: Functional Test-Case generated

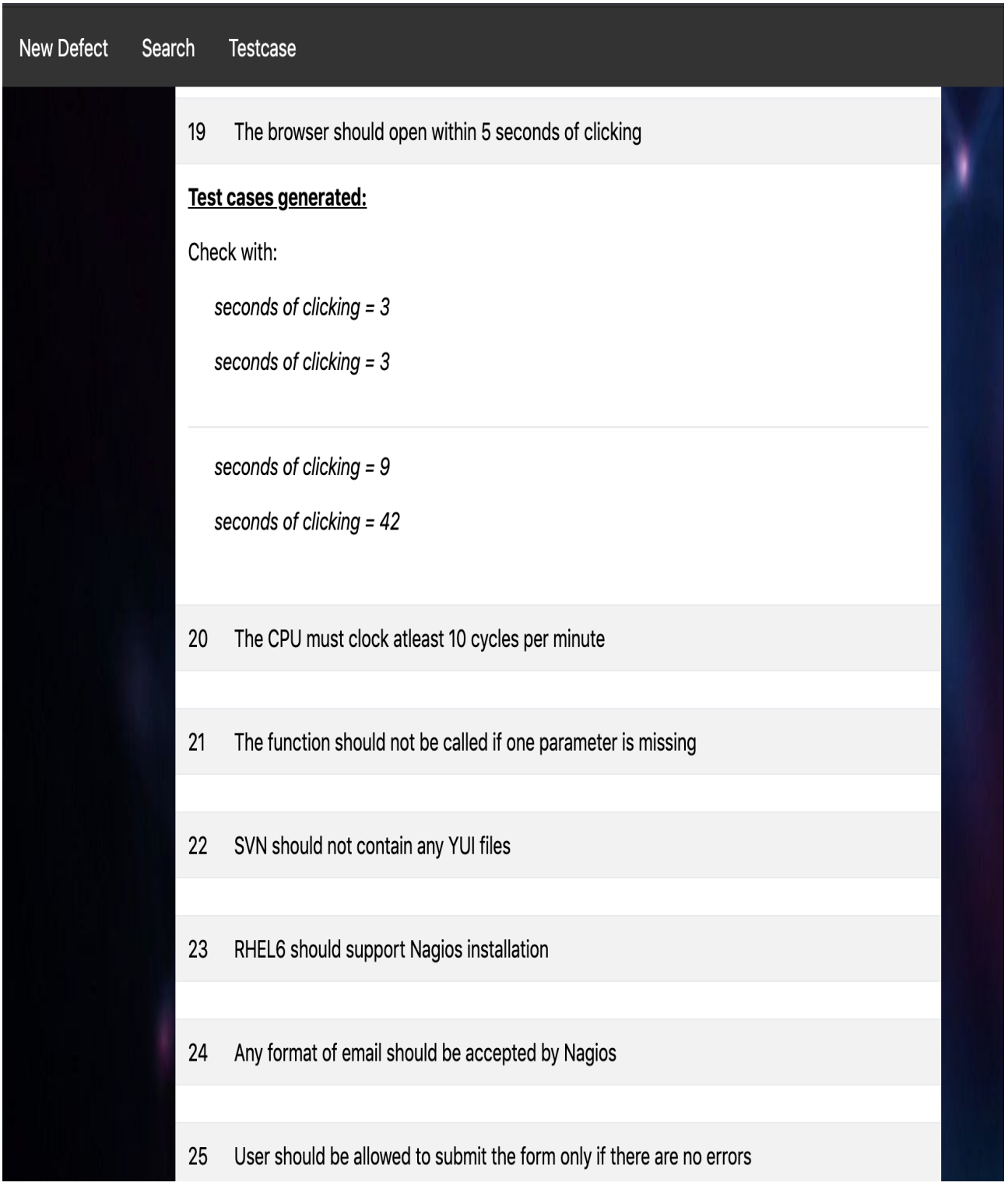
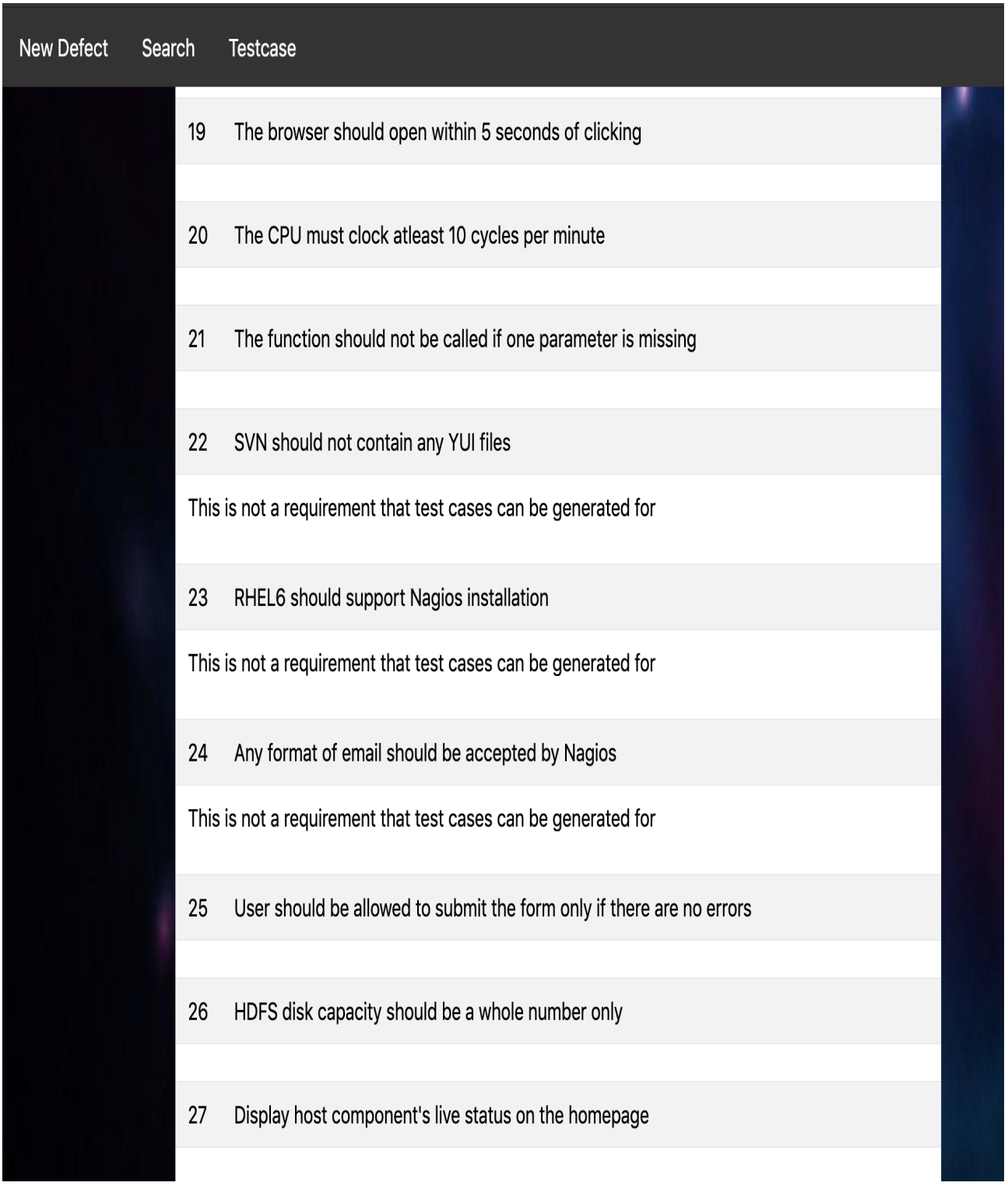


FIGURE 7.9: Identified incapibility



CHAPTER 8

CONCLUSION AND FUTURE WORK

This thesis provides a extensive report on 4 deliverables that have been satisfied by the system namely:

- Given a database of open issues, identify issues similar to that which the user is trying to submit
- Given a database of requirements, identify the requirement unsatiated by the issue that the user is trying to submit
- Corresponding to project parameters, predict the criticality of the issue that the user is trying to submit
- Given a requirement, generate test cases for it

The system uses a DAN model over an LSTM model mainly because DAN's perform better in scenarios where the training sentences and testing sentences vary in structure. This is highly relevant to this project as there is a high probability that it will be employed in highly technical scenarios with a diverse input corpus.

The system also uses Rasa, which enables the developers of different projects to define their own levels of criticality, and associate to these, keywords relevant to their project. This also has an added advantage of a very small training time required.

The combination of Sentence Dependency parsing and SMT, provides a powerful tool which can take in multiple related constraints and still provide relevant test cases. However, the system is currently limited to generating test cases for functional requirements which have numeric dependencies only.

Development of a module which generates test cases for non-functional requirements as well is quintessential and will ensure maximum usability and reach of the product.

REFERENCES

1. Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. CoRR, abs/1207.0580.

2. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1).

3. Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, Hal Daume III. 2015. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).

4. Cer, D., Yang, Y., Kong, S.Y., Hua, N., Limtiaco, N., John, R.S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C. and Sung, Y.H., 2018. Universal sentence encoder. arXiv preprint arXiv:1803.11175.

5. De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008.

6. Mikawa, Kenta, Takashi Ishida, and Masayuki Goto. "A proposal of extended cosine measure for distance metric learning in text classification." 2011 IEEE International Conference on Systems, Man, and Cybernetics. IEEE, 2011.

7. Mueller, Jonas, and Aditya Thyagarajan. "Siamese recurrent architectures for learning sentence similarity." thirtieth AAAI conference on artificial intelligence. 2016.

8. Bocklisch, Tom, et al. ”Rasa: Open source language understanding and dialogue management.” arXiv preprint arXiv:1712.05181 (2017).

9. Marcus, Mitchell, Beatrice Santorini, and Mary Ann Marcinkiewicz. ”Building a large annotated corpus of English: The Penn Treebank.” (1993).

10. Manning, Christopher D. ”Part-of-speech tagging from 97% to 100%: is it time for some linguistics?.” International conference on intelligent text processing and computational linguistics. Springer, Berlin, Heidelberg, 2011.

11. Schank, Roger C. ”Conceptual dependency: A theory of natural language understanding.” Cognitive psychology 3.4 (1972): 552-631.

12. Choi, Jinho D., Joel Tetreault, and Amanda Stent. ”It depends: Dependency parser comparison using a web-based evaluation tool.” Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). 2015.

13. Dwarakanath, Anurag, and Shubhashis Sengupta. ”Litmus: Generation of test cases from functional requirements in natural language.” International Conference on Application of Natural Language to Information Systems. Springer, Berlin, Heidelberg, 2012.

14. Myers, Glenford J., Corey Sandler, and Tom Badgett. The art of software testing. John Wiley and Sons, 2011.