5-2022

A Co-Evolutionary Approach to Test Case Generation for Safety-Critical Systems

Brad Thomas Costa

A Co-Evolutionary Approach to Test Case Generation for Safety-Critical Systems

by

Brad Thomas Costa

Bachelor of Science
in
Computer Science
Department of Computer Science
College of Computer Engineering and Science
University of Central Florida
2015

A thesis
submitted to the College of Engineering and Science
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2022

We the undersigned committee
hereby approve the attached thesis

A Co-Evolutionary Approach to Test Case Generation for Safety-Critical Systems by

Brad Thomas Costa

Thomas C Eskridge, Ph.D.
Associate Professor
Computer Engineering and Science
Committee Chair

Ryan White, Ph.D.
Assistant Professor
Computer Engineering and Science
Outside Committee Member

Siddartha Bhattacharyya, Ph.D.
Associate Professor
Computer Engineering and Science
Committee Member

Khaled Slhoub, Ph.D.
Assistant Professor
Computer Engineering and Science
Committee Member

Philip Bernhard, Ph.D.
Associate Professor and Department Head
Computer Engineering and Science

Abstract

Title:

A Co-Evolutionary Approach to Test Case Generation for Safety-Critical Systems

Author:

Brad Thomas Costa

Major Advisor:

Thomas C Eskridge, Ph.D.

Safety-critical software development is a costly and time-consuming process that involves thousands of hours dedicated to test development. Tests must meet stringent developmental guidelines to verify the correct and complete implementation of their parent requirements. Further compounding any such effort is the tendency towards requirement churn or the frequent change to the software and other system requirements. This thesis presents a solution, PyTcGen, that alleviates these challenges by processing natural language requirements and programmatically generating the requisite test cases to ensure the software meets all of the conditions of that requirement. The solution uses template matching to marry requirements to the code that generates tests. This template matching approach adds a further advantage in the form of a co-evolutionary relationship between requirement authors and the system that drives the creation of more concise requirements while simultaneously increasing the usability of the system.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I'd like to acknowledge the considerable contributions of my graduate advisor, Dr. Tom Eskridge. Without whom, I would not have been able to complete this project. His endless patience and tolerance for inane questions make him the consummate educator and scholar. I also want to acknowledge the morale support, time considerations, and accommodations from my wife, Lori Costa, Esq. Without her encouragement and support, I would not have resumed even my undergraduate studies. She supported me while working as an attorney, being pregnant, raising small children, and bouncing between doctors looking for a diagnosis. There is no comparison to her strength.

Dedication

This thesis is dedicated to my wife and sons, who are my world, to my mother, who is my biggest fan, and, even though he will never read it, to my father, the most intelligent man I've ever known. Thank you for supporting me and, even more so, for putting up with me.

Chapter 1

Introduction

In the experience of the author, software testing strategies can be divided into two categories: safety-critical testing or non-safety-critical testing. Safety-critical software is any software for which a bug or failure may result in significant loss, severe injury or death. Safety-critical software development and test are significantly more formalized in their methods than non-safety-critical development and test efforts. Non-safety-critical testing may be formalized but it is not bound by regulations and industry standards to be so. Non-safety-critical testing can be as simple as unit testing written and executed by the developer of the code or it can be extensive programs involving functional, system, integration, A/B, acceptance, and many more forms of testing.

In safety-critical software development, system behaviors are frequently defined via text-based requirements. Non-safety-critical software development may also include text-based requirements but what is different is that in non-safety-critical software requirements can take many forms such as lists of features, use cases, or activity diagrams, to name a few. Safety-critical text-based requirements, however, take the form of discrete statements. Such requirements are often referred to as "shall" statements due to the practice of each requirement containing the word "shall." For example:

"When `Input_1` is > 5 TheSoftware **shall** set `Output_1` to On." (1.1)

Historically, the use of "shall" comes from Government contracting. Ambiguous wording in early software contracts led to disagreements as to what contractually constituted completed software. Later contracts adopted a specific set of terms that were used in requirement documents. These included shall, will, and should, among others. The word "shall" was used to denote items the software had to meet or adhere to in order to be considered contractually complete. Whereas "will" denoted capabilities the software would eventually have to meet. "Will" requirements were meant as a way of guiding design to accommodate future features or modifications. "Should' requirements constituted requests in an "if there is time," or "if it is possible" fashion.

Safety-critical software development involves different techniques than non-safety-critical development requires. Depending on applicable standards, these can include detailed multi-leveled requirement specification with parent-to-child requirement tracing, a one-to-many relationship between requirement and tests, specification of separate test cases and test procedures, disabling of compiler optimizations, source-to-object analysis, off- and on-target code coverage collection, test author and executor independence, and independent-party-observed formal test execution. These processes can be both time-consuming and expensive.

1.1 Background

In safety-critical software engineering, hierarchical formal requirement statements specify software operation. The hierarchy is established in the way requirements are written, linked, and traced. The formality stems from two factors. First, the aforementioned use of the word "shall" and the grammatical structures that support its use. And second, their clear delineation from supporting and explanatory text. While

"shall" statements can certainly be used in non-safety-critical software engineering it is more typical to find that requirements take the form of standard English statements. This can make it difficult to know what it is a true requirement and what is supporting information.

In the hierarchical approach, requirements are translated from customer or product requirements into system requirements and then further into high-level software and hardware requirements, followed by low-level software and hardware requirements. (See Figure 1.1) Hierarchical models for requirements can vary in the number of layers, the specific terms used for each layer, and the categorization. It can be beneficial to group requirements by types, such as functional, performance, etc., rather than hardware or software. What is requisite is that tracing is maintained between the layers regardless of the hierarchy model chosen. Starting from the customer/product requirement layer, requirements decrease in abstraction and increase in specificity. Requirements can also be categorized based on what they are stipulating. Requirements can be functional or non-functional. They can also place constraints on performance, tolerance, or behaviors. A requirement that specifies the color of a button on a screen would be an

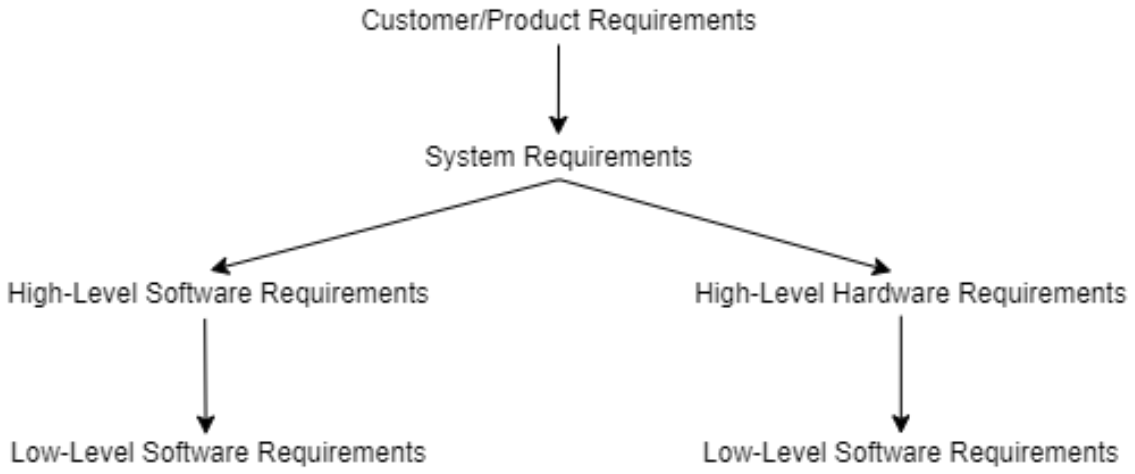

Figure 1.1: Example Requirement Hierarchy

example of a non-functional requirement. One that specifies the supported operational temperature range of hardware would be a performance requirement. Appropriate testing is performed for all types of requirements at each layer. This is illustrated in the "V-model." (See Figure 1.2)

Safety-critical projects tend to have similar characteristics. Project types include transportation: automobiles, trains, and aircraft; medical devices: radiation sources, surgical robots, and medication drips. The mechanical and physical characteristics of such projects mean that safety-critical software applications often have very similar types of coding structures, i.e., state machines, Boolean logic gates, range defined values, etc. For example, aircraft, automobile, and train software must ingest sensor data that gives information about the craft's speed, direction, heading, etc. The software must then perform calculations on those data, make determinations of the state of the

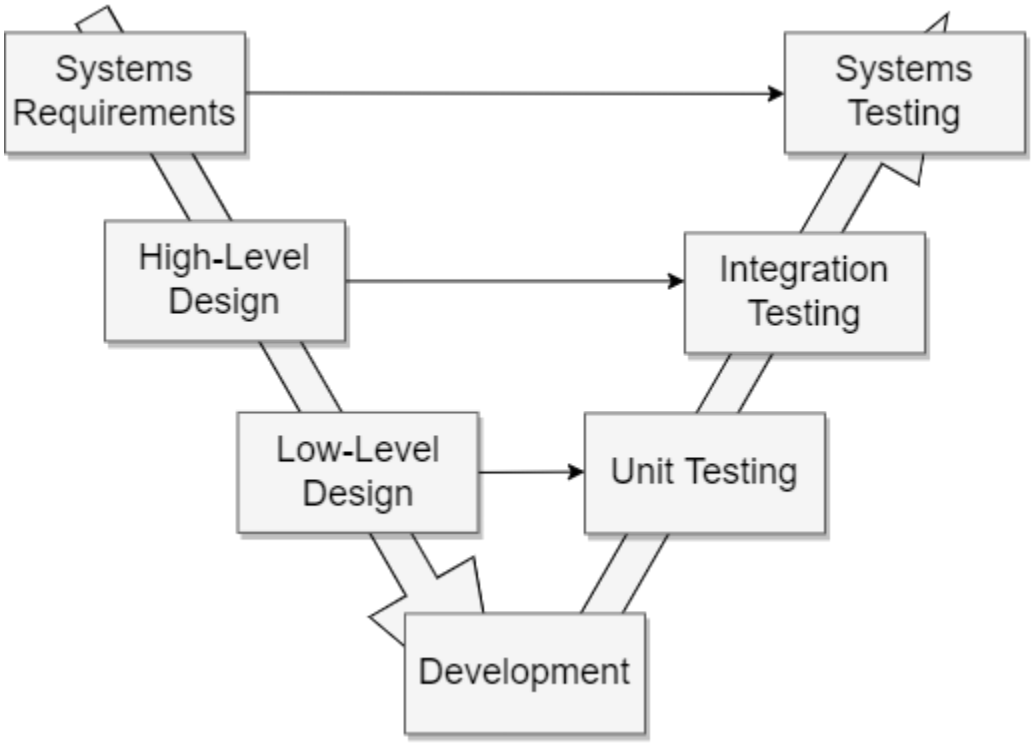

Figure 1.2: Development V-Model

craft, and command actuators. This pattern of reading, calculating, and actuating drives similarities between safety-critical software projects. While software testing in general encompasses numerous different paradigms, the solution proposed in this thesis utilizes this similarity across safety-critical projects to significantly reduce testing overhead.

Safety-critical software imposes a burden on the tester that test designs must demonstrate unequivocally that the software is implemented and performs as defined in the requirements. As an example, DO-178C, the current standard by which the Federal Aviation Administration (FAA) mandates software on aircraft be certified, requires that testing demonstrate that, for requires that testing demonstrate that, for requirements that can be represented as Boolean logic expressions, each input must be shown to independently affect the result of the expression [36]. To understand what this implies, let us use an example requirement. This requirement is also used in obtaining experimental results.

> "When AirGroundStatus is OnGround and GroundSpeed is greater than or equal to 120 NMI[1] then TheSystem **shall** set AirGroundStatus to InAir and AirGroundStatusOverride to True." (1.2)

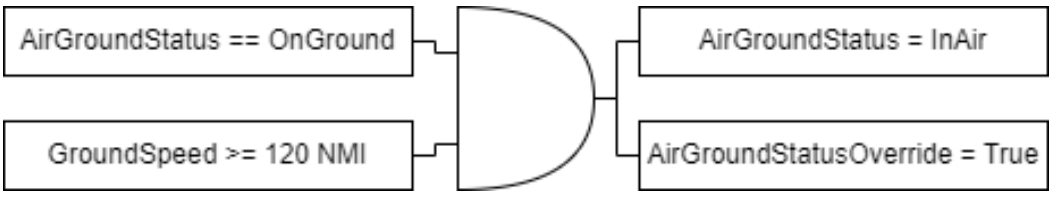

Figure 1.3: Requirement (1.2) Logic Diagram

[1]Nautical Miles

In this example, Requirement (1.2) can be visually represented as the logic diagram in Figure 1.3. From the text of the requirement and the ability to represent that text as a logic diagram, it can be seen that this example requirement can be represented as a Boolean expression. From the diagram let the expressions

"AirGroundStatus == OnGround"

And

"GroundSpeed >= 120 NMI"

equal inputs "A" and "B," respectively. Then all possible values for the results of the logic expressed in Requirement (1.2) are shown in Table 1.1. Note that the number of rows in the table is directly proportional to the number of Boolean inputs to the gate. That is, the possible combinations of inputs for a Boolean logic gate are 2^n, where n is the number of inputs. However, as expressed by DO178C, a test author does not have to produce 2^n test cases in order to "cover" this requirement. Instead, only each input's ability to independently drive the result must be shown. Continuing with the example of the AND gate with inputs "A" and "B," consider the case when both inputs are *True*. This is the only instance when the gate will evaluate to *True*. Thinking of this case as the default or "positive path" case, testing this case serves as the base to prove each input's effect on the gate. That is, to show an input's effect on the outcome of an AND gate we, therefore, must show that when a particular input is *False* the gate is *False*. In the two-input "A/B" example this is demonstrated with the cases where A is *False* and B is *True* and where A is *True* and B is *False*. Since the case when both inputs are *True* establishes that if both inputs are *True* than the gate is *True* it follows that in those cases where only one of the inputs is *False* it is that input that is "driving" the gate's output to be *False*. This process is the same for OR

or XOR gates with the caveat that the positive path of the gate is the case where all inputs are *False* and each input "drives" the gate when it is the only *True* input value.

Table 1.1: 2-Input AND Gate

A	B	Result
T	T	T
F	T	F
T	F	F
F	F	F

To show that this requirement has been implemented correctly in code, using the fact that the requirement is an AND requirement, a tester needs to check only three of the four rows shown in Table 1.1. Those are:

Table 1.2: 2-Input AND Gate Test Cases

A	B	Result
T	T	T
F	T	F
T	F	F

To understand why one needs notice two things. First, as an AND comparison, a false value for any input will result in the expression/gate evaluating to *false*. Second, in the fourth case in Table 1.1 FF, it is impossible to know which of the two false values drove the result of the comparison to false. Demonstrating an inputs independent affect on an output accomplishes two main things. First, it proves that the input is utilized as entered/read. And second, when taken together with all cases for a given express, it proves the gate is implemented as specified in the requirement in the minimum possible number of test cases. Therefore, a test with two false input values would provide no new information beyond executing tests where the input values are FT and TF. Recall from above that the intent is to demonstrate each input's affect on the output. A FF input value set does not demonstrate which input drove the gate *False*. Here it can be noted that due to the design of most modern programming languages and Arithmetic

Logic Units (ALUs), the evaluation of an AND expression will halt and evaluate to false as soon as the first false in the input chain is encountered. Thus, further obfuscating the FT and FF cases. Extrapolating to the general case, this example shows that the number of tests required to demonstrate the independent effect of each input of a Boolean expression is $n + 1$ rather than 2^n thereby dramatically reducing the number of input combinations required.

Requirement (1.2) can also be used to illustrate another fundamental aspect of safety-critical testing, equivalence classes. Note that in Requirement (1.2),the input labeled as B is an inequality function. As a result, testers must also design their tests such that the function is shown to be correctly implemented in code. This is done through knowledge and understanding of equivalence classes. From set theory, we know that equivalence classes are defined as follows:

Definition: Let R be an equivalence relation on the set A, and let $a \in A$. The

equivalence class of a under the equivalence R is the set

$$[a]_R = x \in A | xRa$$

of all elements of A which are equivalent to a. [13]

In the example, input B can be expressed as:

$$f(x) = \begin{cases} true & x >= 120 \\ false & x < 120 \end{cases} \tag{1.3}$$

where x is the input 'GroundSpeed.' Note that if a tester were to use this definition directly, regardless that it is correct, in its current form it conceals a required test. A more correct way of expressing B would be:

$$f(x) = \begin{cases} true & x = 120 \\ true & x > 120 \\ false & x < 120 \end{cases} \tag{1.4}$$

With this representation, it is easier to see that in order to cover this requirement a tester must generate at least three tests. Similarly, if the requirement is represented as in Figure 1.4 below:

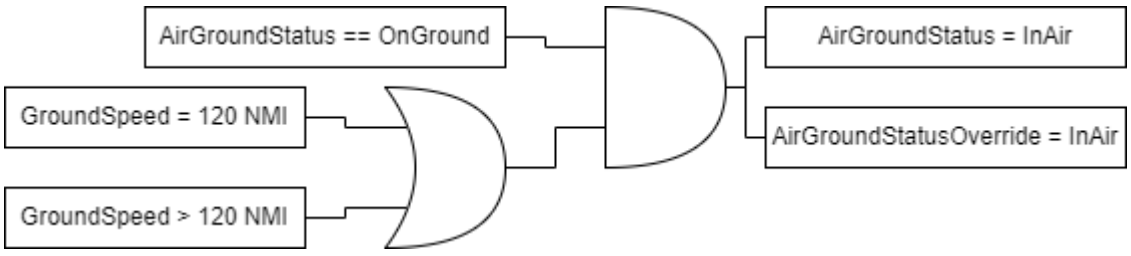

Figure 1.4: Requirement (1.2) Logic Diagram 2

one can see that second input is itself a 2-input gate. In this case, an OR gate. And corresponding to Equation 1.4 the number of cases required to cover a 2-input gate is

3. However, note that there are infinite values that satisfy the inequality $x > 120$. So which value should be chosen? The meaning of an equivalence class is that each value within it, no matter the size of the set, is equivalent to every other. This would imply that in testing the input B relation, choosing 121 as the value to test or 2,147,483,647 would not produce different results. This is correct in terms of equivalence class coverage but recall that the objective in test generation is not simply equivalence class coverage but instead to prove the implementation of Requirement (1.2) in code through the use of equivalence classes. Equivalence class coverage itself will not provide the implementation of the requirement to be correct. This is an important distinction because, while both values are valid for testing purposes, the value 121 can provide more information about the implementation than the other. The reason for this is precision.

Assume for a moment that the system being developed in this example uses only integer values. This is a common occurrence in safety-critical software due to rounding errors that can occur with floating-point numbers [39]. This means that the smallest differences from 120 representable in the system are 119 and 121. Knowing this allows a tester to distinguish between the values in the equivalence class generated by the function $x > 120$ because any value in the class *other than* 121 allows for a value to be used in the source code that is not 120. E.g., if the tester were to chose 150 as the value for the test then the developer could have used any number in the range $(-\infty, 149]$ as the threshold value in the greater than condition and the test would still pass. But, if 121 is chosen and the results taken along with the results from the similar tests developed for the other two equivalence classes generated by the inequality, those being 119 and 120, then it is a correct statement to say that the condition, as stated in the requirement, has been implemented in code. To illustrate this a different way consider the following scenario. The requirement is the same and states the condition as $x > 120$ but the when implemented in the code that threshold value is entered as

121. When tested, if the value used was unrestricted and 122 was chosen as the test value, the test would pass and the error of the condition being entered as > 121 rather than the required > 120, would be unrevealed.

This is partially illustrative of the final two parts to understanding the basics of safety-critical testing as background for the ideas presented in this paper. I.e., boundaries and ranges. The threshold of 120 in Requirement (1.2) is an example of a boundary and choosing the values {119, 120, 121} for testing is an example of proper boundary coverage. To understand test coverage of ranges the ideas of normal- and robustness-range values must be introduced. (Note: while it may be more grammatically correct to label them robust-range values it is industry standard to refer to them as robustness values.) Consider the number line in Figure 1.5. This figure represents some range [n,m] where n is the minimum value and $m > n$. The green brackets represent what is considered the normal range values. The values outside the normal range, represented by the red brackets, $(-\infty, n-1], [m+1, \infty)$ are the out of normal range values, with n and m the boundary values.

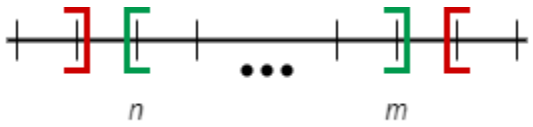

Figure 1.5: Range Number Line

Normal values can be thought of as those values that the system is expected to encounter under nominal operating conditions and those that the system designers do not expect the system to encounter under normal conditions are the robustness values. An example that may be more familiar for some readers would be that of a phone number entry on a web form or application GUI. The normal values might be a set similar to {0-9, ".", "-", "(", ")", "+"} but a user may accidentally or with malicious intent enter other characters such as "@" or "q." These are examples of values that do

not fall within the normal range and are therefore robustness values.

Applying these concepts to Requirement (1.2), notice that the name of input B is "GroundSpeed." For the sake of argument assume that the ICD that is associated with our hypothetical requirement specification for Requirement (1.2) defines the valid (or normal) range for the GroundSpeed as [0, 1984] and the units as NMI/h (nautical miles per hour). 0 for obvious reasons and 1984 or ~Mach 3 because well, if a commercial airliner is doing Mach 3 then there are more pressing concerns than ground speed. The robustness values for this normal range are then the ranges $(-\infty, -1]$ and $[1985, \infty)$. Therefore when developing test cases that involve the range of values for the input GroundSpeed a tester must consider the set of values $\{(-\infty, -1], [0, 1984], [1985, \infty)\}$. Conveniently though, each range in the set can be considered an equivalence class with one notable caveat, zero. To illustrate, imagine another input whose range is defined as [-10, 10]. Under the normal/robustness range and boundary value rules described above the test values associated with this range and the upper and lower boundaries would be {-11,-10,-9,9,10,11}. But note that [-10, 10] includes 0 yet 0 is not included in the set of test values. For many test scenarios, this may be of no consequence. However, when dealing with computers zero can be a troublesome value as it can cause a program to crash if the value is encountered at certain points in program execution, such as in the denominator of a division operation. In applications like aircraft control where formulas are used frequently in code to calculate useful values like groundspeed and airspeed, it is not uncommon to find division operations where the divisor is itself an input or is a variable whose value can be driven by inputs to be 0. Thus, including 0 in the set of values used to derive tests can expose unprotected arithmetic operations in safety-critical source code. Additionally, when programming languages that include referential structures like pointers are used, using 0 in tests that might not otherwise require it can expose improper referencing/dereferencing of pointers. Thus, though not

required, it is good practice to include 0 whenever possible[2].

1.2 Problem Description

From Section1.1 it can be seen that any safety-critical software development project will require a significant effort and investment in test development. What is not clear from that discussion are the compounding and complicating factors that safety-critical projects face. Test development is often assigned to younger engineers as a way to introduce them to safety-critical engineering or may be outsourced completely. Requirements development is often made more difficult by the ambiguity inherent in written communication. Other requirement development difficulties are best illustrated with the quote: "The strongest human emotion is neither love nor is it hate. It is the need to change someone else's text. *-a safety-critical proverb, source unknown*" This is a colorful way of noting that frequently the comments made during a peer review are not critical of the content of a requirement but rather the way it is worded. This highlights another significant impact on safety-critical testing scope, Requirements Turn or Churn. This is the name given to the problem of requirements frequently undergoing updates during the course of the development process. As requirements are implemented in code, developers find previous assumptions will not work and requirements are updated. Gaps in requirements are identified as well and new requirements added. Incremental testing efforts reveal problems in the design and requirements are added, deleted, or changed to compensate. Problems facing safety-critical test development efforts can therefore be summarized as follows:

---

[2]There are other such special values such as NAN, INF, and -INF as well as the minimum and maximum values supported by the data types used but, the impact and applicability to testing for these values are much more specific to the requirement in question and/or application being developed and are thus omitted for brevity here.

1. Safety-critical test development requires an understanding of Boolean logic expressions, equivalence class relationships, range and boundary conditions, and the ability to apply that understanding to complex natural language requirements.

2. Test development for even experienced engineers can be a very time-consuming process that can add orders of magnitude to the time and cost of a project's development.

3. Frequent changes in parent requirements drive frequent redevelopment and execution of tests.

1.3 Proposed Solution

In this thesis we propose a system that addresses the problems enumerated in § 1.2 in the following ways:

1. Automatic translation of natural language requirements into object-oriented structures via requirement templates (see § 3.2).

2. Automatic generation of tests for Boolean logic expressions, inequalities, and ranges (see § 3.3).

3. A co-evolutionary process of the language for writing requirements and the development of templates used to parse and generate test cases.

4. A modular architecture for the rapid adoption new technologies or alternative approaches.

The proposed system (see Figure 1.6), called PyTcGen for Python-based Test Case Generator, requires three sets of inputs. The requirements to be parsed, the set of

requirement templates against which the requirements are to be matched, and Python-based code that is to be executed for the associated template (see § 3.2.2). The code ingests natural language requirements, tokenizes them utilizing the Stanford CoreNLP [27], and matches the tokenized requirement to a test-case generation template. Templates are associated with Python code at the time they are ingested. If a requirement is successfully matched with a template, the code associated with that template is executed, producing the required tests needed to cover that requirement. If the system is unable to match a requirement with a template the template determined to be the closest (see § 3.2.1) is displayed. This allows the user to determine whether the is a slight variance in the requirement or template that needs to be addressed, additional

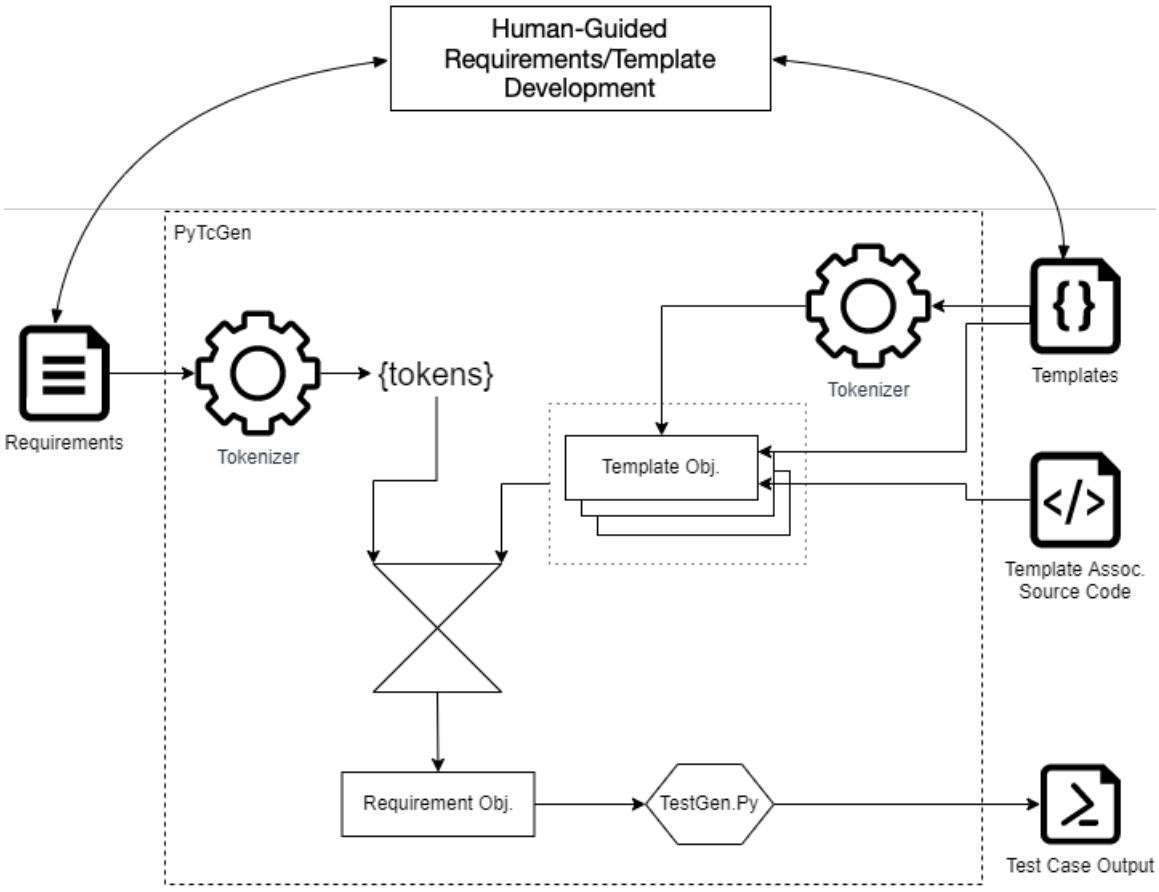

Figure 1.6: PyTcGen Block Diagram

changes are required. This provides system feedback that allows for the co-evolution of the system and the requirement designers that use the system.

1.3.1 Co-evolution of The User and The System

The system is not designed to be static with predetermined templates that force the user into a rigid paradigm for requirement definitions. Rather, it is the intent of the design that an iterative cycle of template and requirement generation develops such that system designers take into account the known-good templates. Likewise, any sufficiently complex and new system will necessitate novel requirement structures, and new templates will be developed to accommodate those requirements. The use of the tool would continue in this fashion across individual project lifespans with each use of the tool increasing its capabilities and its usefulness. At first, the tool would be able to generate test cases for a small fraction of requirements for a given project. For the sake of argument, assume the requirements covered is 30% in this hypothetical scenario. Given the cost and complexity of many safety-critical development efforts, a nearly $\frac{1}{3}$ reduction to test development would be highly desirable by any program/project management involved. Indeed, any reduction in test case development costs would be valuable, because without automation, even the test cases for the simple requirements would need human programming effort to develop. But, as the use of the system continues on the hypothetical project and over additional projects the fraction of requirements whose tests can be automatically generated by the tool will only increase until a limit is reached. The percent of requirements that can be covered utilizing the tool is likely asymptotic with regards to the percentage of software requirements that are functional and define behaviors that can be represented as Boolean logic expressions.

As will be apparent in § 2.2, there are many approaches to test generation, includ-

ing many tools that rely on several different techniques. Methods include model-based systems, controlled natural language inputs, and purpose-designed user interfaces to accept requirement structure and information. What is common to the problem across solutions is that researchers must overcome three challenges. First, researchers must extract structural information from requirements, data such as inputs and outputs, and their relationship. Second, a mechanism must translate the structure of the data extracted from the requirement to that which is applicable to test generation. Approaches here include model-based systems that translate a requirement model into a test or testable model. The third and final challenge is the generation of tests from the extracted and translated information. Our approach takes a modular yet holistic path. Rather than focus on information extraction, translation, or generation individually, all three problems are addressed modularly. While presented here as a Python-based tool, this approach could be applied in any requirement-to-test scenario across any applicable domain using whichever languages and technologies are best suited to its application.

An additional difference of our approach is the applicability of the co-evolution of the user(s) and the system. In our experience, the minimal set of tests that an automated system can generate for a given system is asymptotic with regard to the number of applicable requirements. Put another way, a sufficiently complex system will have other requirement types beyond those supported by this or any proposed solution. Nor is exhaustive testing required to increase the probability of detecting errors in Boolean logic expression implementation [11]. Therefore, we do not claim that this approach can generate 100% of the required tests for any given system. What is claimed is that for any automated test generation system, co-evolution of the system for parsing and generating test cases and the language users specify new requirements in is a necessity. Whether the requirement input method is model-based, a graphical

interface, NLP, or something else, users will have to adapt the requirement structure to the given input method. Therefore, a feedback loop will develop where requirements are maladapted to the system's constraints, feedback is given, and the requirement *or* system is adjusted. In our approach, this feedback loop does not result from exceptions in the application or error messages; instead, it is a feature.

Chapter 2

Literature Review

Any approach to generating tests from natural language inputs requires two primary functional components. First is a mechanism for identifying the crucial entities and values in requirements. Typical solutions to this challenge include NER and restricted languages. The second is a system that translates the associations and structure defined in the input to the tests. Structural translation approaches fall into two categories. Those are model-based approaches and automatic test generation. The following literature review highlights papers applicable to these challenges.

2.1 Entity and Value Identification

Any test generation solution that uses natural language as an input must solve the problem of identifying the key terms and values first. Identification of these terms identifies the"what" in the test generation problem. Take the requirement substring "...when Input1 equals 5...," any proposed solution needs to identify Input1, equals, and 5 as the key terms. Input1 is the I/O variable identifier, 5 is the value, and equals is the relationship between the two. In an idealized system, the solution would

also identify additional properties of each. Input1 would be marked as a numerical input and the associated value 5 as a scalar threshold. Such attribute labeling would allow applications beyond test generation, such as requirement consistency checking and model generation.

2.1.1 Natural Language Processing of Requirements

NLP of requirements has several applications beyond test generation. Ambiguity is an example problem beyond testing. Ambiguity in requirements leads to difficulties in code implementation and testing. In terms of test generation, NLP is a mechanism for identifying the tests' inputs, outputs, and values. What follows in this section is a selection of papers that utilize NLP techniques in ways that are directly or indirectly related to test case generation.

In "Analyzing Quality of Software Requirements; A Comparison Study on NLP Tools," [29] Naeem et al. compare and contrast five tools available at the time of publication, QVscribe, QuOD, Innoslate, RAT, and RQA, where RAT is a new tool proposed by the authors. The tools are compared based on their abilities to detect words or phases the team deems "bad." They are imperatives, "vagues," directives, options, universal quantifiers, and negatives. This qualification of requirement wording has applications in managing the quality of software requirements as improper phrases and poor word choice can contribute to the ambiguity of natural language requirements. Imperative counting is done because having more than one imperative in a requirement can suggest the need for more than one requirement. Vague terms such as "adequate" increase requirement ambiguity. Directives in the context of requirement writing are beneficial. On the other hand, Options are considered bad form in requirement writing as the purpose of a requirement is to provide unequivocal direction. Universal qualifiers such as "most," "always," or "none" increase ambiguity with their lack of specificity.

The term "negatives" is not defined in the text, but the assumption is this refers to requirements that define behaviors in terms of something the system does not do. For instance, a requirement that states, "...shall not cause an error..." They conclude that, although none of the tools investigated were superior to all others across all the features they surveyed, their tool, RAT, is superior because it includes the same feature set as the other tools, plus additional features. And, it performed better in precision and recall in most feature categories.

Shen et al. [37] propose the combined application of deep and active learning to improve Named Entity Recognition (NER) over other approaches. NER in requirements processing is important as it can serve as the basis for other research areas. Including but not limited to automated requirement processing for consistency checking, among different quality measures/improvements, ontological mappings, or model representation from natural language requirements. Robust and accurate NER could augment or replace the template matching system in the proposed solution. Their approach seeks to demonstrate near-peer or better NER performance with other methods while training on smaller data sets. By employing active learning, their team was able to strategically annotate a limited collection of training data compared to traditional deep learning approaches that require massive sets of annotated data to achieve accurate results. By employing this approach, Shen and their team achieved state-of-the-art results in identifying named entities correctly 86.86% of the time in English text and 75.63% in Chinese. These results are significant to the NLP of requirements due to the limited number of requirements typically available to researchers.

Arellano et al [4]. use NLP technologies to extract application-specific ontologies from natural language requirement specifications. Ontologies provide the means to check that all system properties defined within the ontology have corresponding requirements. Checking against an ontology can also detect redundant or contradictory

requirements. Their work includes a prototype software tool that implements the proposed ontological framework approach to a simplified model of an aircraft. They employed the Python NLTK for the NLP processing tasks of their application. Several times in the paper, Arellano et al. mention templates as a means of mapping requirement coverage to originally required capabilities, among other possible applications. Future work proposed by the team includes a formal analysis of property attributes of the ontology by using NLP techniques to extract ontological information directly from requirements.

In the paper from Btoush and Hammad [8], NLP is used to extract entities and their attributes and relationships to build Entity-Relationship (ER) diagrams. ER diagrams are particularly useful in the domain of relational database design. Extracting ER diagrams directly from natural language requirements would have significant applications in database development and related fields. Being able to generate ER diagrams from natural language requirements would also be transferable, in theory, to other model-based design approaches. Btoush and Hammad utilize sentence segmentation, tokenization, part-of-speech tagging, chunking, and parsing combined with domain-specific heuristics to extract the information required to establish entity relationships from the natural language inputs. Unrestricted language use and the NLP tools' inability to understand all relationships and attributes without extensive domain-specific heuristics hampered this approach. They propose future work exploring the use of semantic understanding in NLP as a way to fill the gaps in ER extraction.

Stance detection through NER is the target of research by Küçük [21]. Stance in this context refers to the position an individual takes on a social or political issue. Küçük utilized NER on stance-annotated Turkish tweet data and then included the NER data as Support Vector Machine (SVM) features for stance detection in non-annotated tweet data. Küçük and Dilek repeated this process with SVM that already

included unigrams, again with bigrams as features, and finally with unigrams and the presence of hashtags treated as features. A unigram is an n-gram consisting of a single item from a sequence. In the context of this paper, a unigram is equivalent to a token. The conclusion was that utilizing NER in conjunction with SVM, including unigrams, greatly enhanced the capabilities for determining sentiment in tweet data. Future work planned included the application of this technique in other languages, including English.

Zhao et al. [44] provide a mapping study that surveys the literature regarding Natural Language Processing for Requirements Engineering (NLP4RE). Their survey focuses on the state of the literature, state of empirical research, research focus, tool development, and the usage of NLP technologies. Their survey examined 404 primary source documents covering 36 years of research efforts. Of those, 67.08% were solution proposals. Only 7% of proposed solutions surveyed evaluated the proposed approach in an industrial setting, indicating a clear need for industrial partnerships. 42.70% were characterized as focused on the analysis phase of requirements development. Of the 404 papers surveyed, nine were categorized as applying NLP techniques to the application of test generation.

Lample et al. [22] achieve what they claim are the best results reported for NER in "standard evaluation settings" up to publication. Their approach is novel in that it employs output label dependencies via a conditional random field architecture or a transition-based algorithm that constructs label chunks. They also use pre-trained word representations and character-based representations that capture morphological and orthographic information. Only one model, the team, compared their system against achieved a higher accuracy of 91.2%, but this model required hand-engineered features. In contrast, theirs achieved the highest accuracy of those tested at 90.94% without additional input features.

Yang, Zhang, and Dong propose utilizing recurrent neural networks to learn patterns at the sentence level to improve NER [43]. Identifying patterns at the sentence level would be useful in the proposed solution as NER in natural language requirements is hampered by the lack of large corpora used in training NER solutions. Their approach takes the novel route of replacing named entities in training data with their respective label. For instance, replace a person's proper name with the PER tag. They employed recurrent Long Short-Term Memory (LSTM) and convolutional neural network models in their experiments. Utilizing this approach, the team achieved an accuracy of 91.62%, with the next closest accuracy of those proposed solutions tested being 91.21%.

In their paper, "Software requirements as an application domain for natural language processing," Diamantopoulos et al. propose a method for automating mapping requirements to formal representations utilizing semantic role labeling [14]. The team first obtained and then processed several hundreds of real-world requirements. They then devised an ontology that describes a static view of software systems. After this, the team developed a semantic parsing model that automatically generates ontology-based representations from textual requirements. Their method outperformed models that employ word-level patterns and one that learns syntax-based patterns. The software developed is the basis of their S-CASE platform. The S-CASE platform uses the techniques described in this paper to generate RESTful APIs from natural language requirements. http://s-case.github.io/

Lee et al. attempt to alleviate the need for large labeled data sets in training artificial neural networks in NER through transfer learning [24]. Their work is applied to the domain of patient medical record de-identification. The team employed an LSTM trained on the large Multiparameter Intelligent Monitoring in Intensive Care (MIMIC) datasets. MIMIC is a publicly available dataset developed by the Laboratory

for Computational Physiology that comprises de-identified health data associated with thousands of intensive care unit admissions [34]. The researchers then transferred the weights, and the new model was trained on the i2b2 2014 dataset. Lee et al. then repeated this process for the i2b2 2016 dataset. The i2b2 datasets are collections of de-identified patient discharge summaries [12]. In both cases, it was determined that models trained on the smaller datasets employing models that utilized transfer learning performed significantly better on NER than those models trained only on the smaller datasets.

2.1.2 Restricted Natural Language

One of the difficulties in NLP is the ambiguity in written language. Variances that a human mind finds trivial can mean the difference between success and abject failure for artificial interpretation and understanding mechanisms. Restricted natural languages are a solution designed to reduce the complexity of the input language by restricting the structure and/or vocabulary. Restricted languages reduce the complexity of the input text, but the reduction comes at a price. Adopters of a restricted language approach must train users on its use. This section summarizes some restricted language approaches applicable to test generation.

In their paper, "Automated Formalization of Structured Natural Language Requirements," Giannakopoulou et al. offer a framework that constructs temporal logic formulas from FRETISH, a restricted natural language [17]. FRETISH allows only four fields, *scope*, *condition*, *timing*, and *response*. Each possible combination of field types defines a template with Real-Time Graphical Interval Logic (RTGIL) semantics. The case studies conducted in conjunction with NASA show that requirements tend to fall in a limited number of recurring patterns. Among other future work, the authors are pursuing proof of their formalization algorithms and exploring the use of NLP to

fit natural language requirements into FRETISH statements.

Carvalho et al. propose using the Software Cost Reduction (SCR) syntax as an intermediate layer coupled with *SysReq-CNL* as the input layer as the basis for test generation [9]. Natural language requirements are restricted to the Controlled Natural Language, *SysReq-CNL*. Processing of input requirements is achieved through a multi-layered approach. The first layer is designed to verify the syntactic validity of the input concerning the *SysReq-CNL* language specification. The second layer associates semantic meaning. The third stage generates the SCR specification. The tool T-VEC[1] then generates tests. Evaluation of their approach yielded 100% precision and generated 85% of expected test vectors. While these results are impressive, their system relies on the use of their proposed Controlled Natural Language (CNL) grammar *SysReq-CNL*. In contrast, our approach restricts natural language only insofar as input requirements must match a template.

2.2 Test Generation

As described in § 1.1, safety-critical test generation is a complex problem for computer-based solutions. Systems for test generation fall into two categories, model-based and non-model-based. Model-based systems have been developed to generate tests and eliminate the ambiguity of natural language by eliminating the language. Model application to test generation is a logical extension of its use in replacing natural language requirements. Non-model-based approaches utilize a variety of methods to generate tests. These include cloning, part-of-speech parsing trees, test value randomization, and purpose-built tools. The following subsection detail a selection of model-

[1]T-VEC is a test generation tool developed by T-VEC Technologies.

and non-model-based approaches.

2.2.1 Model-based Approaches to Test Case Generation

Model-based methods to code and test generation have had success in industry. To the point that Model-Based Systems Engineering (MBSE) experience is a common request for systems engineering positions in industry. But model-based approaches have not completely supplanted natural language approaches owing to several factors. Model-based approaches require not only that system designers understand the model rules but also the software engineers, testers, and other stakeholders as well. Additionally, there are competing standards vying for adherents. MBSE tool developers will also include variations on existing standards and offer their own further complicating the landscape of model rules. These factors, plus the monetary costs associated with some of the more popular MBSE tools, also present a barrier to adoption for many. In the proposed solution, code is written for each template. The code serves as the model for the translation structure defined in the template and the information extracted from the natural language requirement into the requirement's tests.

Chatterjee and Johari [10] propose a tool, STATEST, that can formalize informal software requirements and generate tests based on the resultant formal specifications. The tool requires that the user translate informal requirements into use cases and state transition diagrams. The use case specification in the tool consists of standard use case fields such as *actor*, *goal*, and *precondition*, among others. When used as presented, STATEST can generate tests for all entered requirements in IEEE 829 standard format. While an impressive result, their solution differs from ours in that it requires that requirements follow the IEEE 829 format, whereas ours seeks to use unrestricted natural language.

In "Automated Test Case Generation from Domain Specific Models of High-Level

Requirements," Olajubu et al. describe a Model-Based Testing (MBT) approach [32]. In this approach, requirements are represented using a Domain Specific Language (DSL), and Epsilon Generation Language (EGL) serves as the foundation of a template system for model-to-test transformations. The approach represents a work-in-progress as only a single-input Boolean logic operator and range-based inputs were supported. Their approach successfully generated test cases for this limited set of input requirement types.

In 2017 the same authors above, Olajubu et al., continued the exploration of test generation using model transformation techniques [33]. A domain-specific language is applied using XText with the intent being to generate tests to satisfy MC/DC. MC/DC is the abbreviation for Modified Condition/Decision Coverage. MC/DC seeks to ensure that all branching paths of a condition have been exercised and that each Boolean expression's independent effect on the outcome of a condition has been demonstrated. The use of XText allows for a text-based modeling approach to represent high-level requirements. Similar to their earlier work, this paper restricts the input to Boolean logic expressions. Olajubu and their team compared this approach with an earlier proposed system from another author. Both methods generated the test cases required to achieve the MC/DC of the expressions tested in their experimentation.

In her thesis, "Automated Test Case Generation from Domain-Specific High-Level Requirement Models," Oyindamola Olajubu proposes a High-Level Requirement Modelling Language (HRML) [31]. The proposed HRML is a domain-specific language where the domain is those requirements that can be termed software high-level or low-level requirements in the aviation industry. HRML is an Extended Backus-Naur Form language defined in XText. Each requirement defined in HRML can have up to four parts, *Input*, *Output*, *Definition*, and *Behavior*. Inputs and Outputs can have parameters; the others, in turn, have sub-requirement types. Using model transforma-

tion techniques, the entered models serve as the basis for generating tests that satisfy coverage of the input requirements.

In the 2014 report, "Automatic Requirements Specification Extraction from Natural Language (ARSENAL)," Ghosh et al. propose ARSENAL as a tool for generating formal model specifications directly from natural language requirements [16]. ARSENAL uses semantic parsing to extract relationships from natural language requirements. The extracted information is modeled in linear-time temporal logic. The authors suggest that the approach could be ported to additional model types. The authors evaluate ARSENAL using the requirement sets they term as FAA-Isolette (FAA), which defines requirements for a hypothetical infant incubator thermostat, and the SAE standard Time-Triggered Ethernet (TTEthernet) that specifies requirements for a TTE communication platform that is used in space applications [25][19]. They evaluate how well the solution performs by the percentage of requirements ARSENAL can parse without human intervention. On automating the translation of requirements into logical formulas, ARSENAL achieved 95% automation of the FAA requirement corpora. At the same time, on the TTE set, it was able to achieve 78% automation. The authors suggest that at least some of the failures in automation may be due to ARSENAL's inability to handle arbitrary functions.

Fu et al. propose using UML as the basis for safety requirement templates [15]. The authors focus on defining generic safety requirement structural element description templates and generic safety requirement sentence pattern description templates. The former can take one of ten forms, functions, interface, input, output, restriction relationships, environment, state transitions, messages, collaboration, and event sequence. The latter is one of four, where a UML diagram represents each. This paper only proposes the use of UML as the basis for safety requirement templates, but evaluations of the approach's effectiveness are not presented.

Allala et al. propose a model-to-model approach to generate tests from natural language requirements [3]. The authors utilize Stanford CoreNLP [27] coupled with meta-modeling of both requirements and tests. Input natural language requirements are less formalized than in other approaches as they take the form of use cases and user stories. The work focuses on the requirement meta-models and employs CoreNLP to extract key elements from those natural language statements in the requirement meta-models. A transformation engine then converts the input requirement meta-model to a target test case meta-model. They test their approach using fifty use cases and fifty user stories.

Bhattacharyya et al. experimented with translating Architectural Analysis and Description Language (AADL) into the input language of the tool Uppaal to verify quasi-synchronous systems [6]. Uppaal is a tool that supports the modeling, verification, and validation of real-time systems that can be modeled as a network of timed automata. To illustrate their approach, the authors utilize the Pilot Flying System. The authors conclude that Uppaal is well suited to modeling real-time properties of concurrent, asynchronous components. However, the translation of AADL into Uppaal input language presented significant challenges owing to Uppaal's limited ability to model nested sub-components that execute synchronously.

2.2.2 Non-Model-Based Approaches

As evidenced by this paper and those referenced, NLP to generate test cases is an active and broad research topic. To help facilitate research in this area, Ahsan et al. provide a systematic literature review of NLP techniques utilized to generate test cases [1]. The authors restricted their survey to the ACM, IEEE, and Springer literature databases. They selected 16 papers published between 2005 and 2014, inclusive. In these papers, the authors utilized six NLP techniques, and the team found eighteen

tools, where the authors of the papers examined proposed ten, and eight were pre-existing solutions. They categorized the papers based on NLP techniques used to generate tests. For example, one such technique defined by the authors is " *Test Intents to Test Cases* " where the definition is given as "Test Intent as the smallest segment of a requirement sentence that conveys enough information about the purpose of the test." The categories were *Test Value to Test Cases*, *Test Intents to Test Cases*, *Test Input and Scenarios to Test Cases*, *Test Condition to Test Cases*, *Multiple ways to Test Cases*, and *Unevaluated*. *Test Input and Scenarios to Test Cases* contained the most significant number of associated papers at six. The authors listed and classified NLP techniques and tools employed in test generation from natural language requirements. Still, they concluded that a future study is required to provide comparisons and contrasts of those surveyed, including their strengths and weaknesses.

Based on an earlier investigation of open-source projects that found that between 8% and 42% of tests were clones, Landhauber and Tichy propose cloning as a test generation method [23]. In this context, the authors define cloning as, "identifying analogous software components and adapting the test cases of one of the analogs (called the model) to be used on the other (the component under test or CuT)." Their supposition is two-part. First, that analogous functionality between classes functions as an oracle of correctness where the human would function as such in manual test generation. In other words, they are claiming that tests created for one function that has been shown to be correct can be assumed to be correct for an analogous function. Their second supposition was that pairs of analogous classes exist in the target source code. Their solution, *TestCloner*, relies on programmers specifying which methods among classes are analogous. Their proposed approach relies on annotations made in code that their tool, *TestCloner*, utilizes to attempt to match tests from models to other CuTs. Their approach generated 90 tests in evaluation, of which 8 were invalid and 12 revealed

faults.

Verma and Beg use Part-Of-Speech (POS) tagging and the resultant parse trees to generate tests for natural language requirements [41]. Their approach consists of *Preprocessing*, *POS Tagging*, *Parsing*, *Knowledge Representation*, *Merging Graphs*, and finally, *Test Case Generation*. In *Preprocessing*, unwanted words are removed, and multi-word keyphrases are concatenated with hyphens. For *POS Tagging*, Verma and Beg used a POS tagger trained on the Wall Street Journal and the Brown corpus. A shift-reduce maximum entropy parser generated parse trees for each input statement. *Knowledge Representation* is accomplished with ontologies utilizing OWL and RDF. *Merging Graphs* employed a set of simple rules such as matching end nodes. Finally, the authors use boundary value analysis combined with techniques from their earlier work to generate test cases. Evaluating the approach on the single requirement, "The program takes three integer-values from the interval [0,100] and finds whether the equation is quadratic and if the equation is quadratic then find whether roots are real or imaginary or equal," their approach was able to generate 27 test cases when boundary value analysis and when combined with techniques from two of their previous publications, it generated 125 test cases.

Aicherig et al. employ Open Services for Lifecycle Collaboration (OSLC) in a generic framework to generate tests [2]. OSLC is a standard designed for linking disparate systems and software engineering tools via web services. By employing such a standard, the authors hope that others can model their approach to link other requirements, analysis, and test generation tools similarly. To test their approach, the authors use requirements for an industrial airbag control chip. In their evaluation, the authors link SystemCockpit, a requirements management tool, with MoMuT::REQs, a requirement consistency checker, and a test generation tool. By employing OSLC, the authors quickly changed tools at any stage in the framework pipeline.

Singh, Sinhrova, and Bhatia provide a review of test generation tools [38]. Their survey covered sixty tools. The tools were categorized based on *Testing Types*, *Language Support Type*, and *Input*. The authors also indicated whether a given tool was available at the time of writing, the license it fell under, and the techniques or methods used to generate tests. Of the sixty tools, the overwhelming majority were for object-orient languages at 91.66%. Most were open-source or academic, with only 0.25% being commercial tools. The model-based vs. code-based distinction was much closer, with 43.33% and 51.66% shares.

2.3 Our Approach

In the proposed solution (see chapter 3), we combine a system for matching templates to requirements and connect the knowledge the template-to-requirement association creates to the generation of tests via code that is written to match the pattern described by the template. The following subsections highlight literature that applies to template matching (a.k.a pattern matching) and generating tests from templates.

2.3.1 Requirement Templates (Patterns)

In their paper, "On systematically building a controlled natural language for functional requirements," Veizaga et al. present a systematic approach to constructing controlled natural languages [40]. Their paper presents a controlled natural language, Rimay, developed in this manner for defining functional requirements in the financial domain. The authors suggest indirectly that patterns and controlled natural languages are similar in that both restrict the syntax of requirement text.

Yahya et al. examine software requirements from a security perspective through a literature review of nine papers focused on security patterns in requirements [42].

The authors seek to identify requirement patterns relevant to software security to aid requirement engineers who may not be well versed in security considerations. The authors note that there is no automated process by which textual requirements can be mapped to security patterns.

In their approach, Hoffman et al. apply research in trust to develop software requirement patterns designed to increase user trust in information systems [18]. Hoffman and their team built their requirement patterns based on antecedents. Where antecedents are defined as "factors or elements that increase trust." Antecedents are treated as goals in the requirement pattern template used. An example from their paper is the goal, "Satisfy the user need of having a system that provides insights about its inner working." The accompanying template is given as, "The system shall explicitly display or say that it will act in a particular way." Five trust antecedents are addressed in this way. They are "Explication of Intention" (the example provided here), Understandability, Transparency, "Information Accuracy," and Personalization.

In their mapping study, Barros-Justo et al. survey the literature to find evidence of the application of software patterns and requirements engineering in real-world engineering activities [5]. The authors selected twenty-two papers out of 3,070 works. 327 software and requirements engineering patterns were found in the literature they surveyed. Barros-Justo and their team concluded that patterns positively impacted the requirements engineering process and helped novice developers. These results are significant to our work as it implies that the use of templates (patterns) is likely to have benefits in the requirements and software development phases of a project life-cycle, not just the increase in productivity from automatic test generation.

Poza et al. propose applying genetic algorithms coupled with a separate-and-conquer strategy that they claim outperforms existing methods for pattern generation by 233% [35]. Their work has significant implications for our proposed solution as we

see one of the most significant obstacles to adoption being the initial creation of a corpus of templates. One possible solution is that the tool is first employed on a new project, and requirements and templates are developed in tandem. Another solution is that templates are created from an existing completed set of requirements. The work of Poza et al. offers what is likely a far more efficient and elegant solution in that their approach could be applied to an existing corpus of requirements to generate a set of templates programmatically.

2.3.2 Template to Test

Kudo et al. propose the Software Pattern MetaModel (SoPaMM) to address the gap between applying patterns in requirements and pattern use in other phases of software development [20]. SoPaMM can be used to represent both requirement and test patterns. Patterns take the form of Requirement Pattern (RP) and Test Pattern (TP) classes. The classes Functional Requirement Pattern (FRP) and Acceptance Test Pattern (ATP) then inherit from these. The authors demonstrate their approach with the development of a simple web application with user authentication. An example of FRP and ATP linkage is given with the class instances *UnsuccessfulLogin:Scenario* and *UnsuccessfulLogin:TestCase* sharing the same class relationship with *UnsuccessfulLogin:Steps*. The authors use this approach to successfully execute acceptance testing of the eighteen steps defined in their FRPs.

In "Transition Algebra for Software Testing," Liu, Li, and Miao propose "Transition Algebra" as a mechanism for translating Extended Regular Expressions (EREs) into tests [26]. The authors use EREs to model software behaviors and build Transition Algebra as an extension of Kleene Algebra. The proposed Transition Algebra provides the operations *Concatenation*, *Choice*, *Kleene Closure*, *Range Closure*, *Limited Output*, *Interleaving*, *Parallel*, and *Constraint*. The authors offer proofs for their

algebra operations and five examples of its application in testing multi-threaded, parallel, probabilistic, sorting, and state machine applications. Their work demonstrates the feasibility of pattern (EREs in this case) translation to test.

Nebut and their team explore the application of test patterns based on software product families and derived from functional requirements specified in UML in their paper, "Automated Requirements-based Generation of Test Cases for Product Families" [30]. The authors refer to these test patterns as *behavioral test patterns* or behTP. A behTP consists of three parts: an accept scenario which represents the specific behavior the test designer wants to test; reject scenarios which are the behaviors the test designer wishes to avoid testing; and a prefix which is the behavior needed to place the system under test into the state required to execute the test. BehTPs are generic and include wildcards and variables to allow for their application to the requirements specific to the system under test. The authors evaluate their approach on a product codebase where 9% of the code is known to be dead and approximately 26% is robustness code. They generated 18 behTP and measured the percentage of code covered by the resultant tests. They found that their test execution covered 98% of nominal and 50% of robustness code. They hypothesize that building behTP specifically for robustness would improve the disparity between the two. Their findings are relevant to our work because their approach has similarities to ours. In our approach, requirement templates contain variables that identify the key terms and values needed to translate the requirement structure into tests.

Moitra et al. propose the tool "Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests," or ASSERT [28]. The authors developed the tool under the auspices of GE Global Research and in collaboration with GE Aviation Systems. ASSERT accepts requirements defined in a structured natural language (SNL). A requirement consists of three parts in the SNL used: a unique identifier, a

conclusion, and a condition. ASSERT includes what the team refers to as the Automatic Test Generation (ATG) tool. ATG analyzes requirements in ASSERT using predefined test coverage criteria based on DO-178C guidelines [36]. As a result of this basis, tests generated by ATG are similar in structure to those generated by our solution (see § 3.3). The work presented by Moitra et al. further demonstrates the feasibility of the pattern-to-test approach.

Chapter 3

Proposed Solution

From survey studies such as that provided by Naeem et al., which surveys NLP tools for requirements processing, and that of Ahsa et al., which investigated NLP tools for test generation from natural language requirements, it is clear there has been an interest in both NLP of requirements in general and for test generation [29][1]. Also clear is that approaches to test generation have fallen into a few broad categories. These are ontological mapping approaches such as in [17], restricted natural language as in [9], and model-based approaches such as is in [9]. Each approach suffers from one or more limitations. Ontological techniques require robust and extensive domain-specific knowledge that must be translated into ontologies. Model-based systems can require extensive training before engineers can correctly and efficiently utilize the proposed system of models. Restricting natural language suffers from a similar limitation in that requirement authors and readers must learn how to express their intentions using the restricted language set and may be hampered by those restrictions if the designers of the language restriction are not careful. The limitations thus far inherent in these approaches lead us to pursue a solution that utilizes templates as a way to avoid restricted languages and models while also providing structure translation scaffolding

from requirements to tests.

3.1 System Description

Recall the block diagram 1.6. The proposed PyTcGen solution works by accepting natural language requirements in a text file with one requirement per line, as illustrated in the diagram by the text file block on the left-hand side. Each requirement is tokenized using CoreNLP. Following this requirement, templates are read and tokenized utilizing the Python NLTK (diagram right-hand side). The last piece of input, the test-generative code for each template, is read in. The template and test-generative code files are one template/code set per line. PyTcGen expects that the lines in the test-generative code file are in the same order as those in the template file. I.e., the code on the first line corresponds to the template on the first line, and so on. When the three input file types are read in, PyTcGen marries requirements to templates via the algorithm described in § 3.2.1. Following successful requirement-to-template matching, the test-generative code for each template that is now married to a requirement is executed, generating a test for the requirement by the algorithms described in § 3.3.

One can summarize this process with the following steps:

1. Requirements are read in and tokenized.
2. Templates are read in and tokenized.
3. Test-generative code is read in and associated with the applicable template.
4. Requirements and templates are matched.
5. Test-generative code is executed for each requirement.
6. If a requirement is not matched with a template, the closest template found is displayed.

PyTcGen is comprised of two major components. The first is the classes and methods contained in TestGen.py. The classes Op and GateType inherit from Enum and only contain operation and gate-type identifiers. The second major component is the classes and methods contained in Template.py. Figure 3.1 is a UML representation of the classes in PyTcGen. The following subsections explain the key algorithms of the proposed solution in more detail.

3.2 Requirement Templates

PyTcGen uses templates to associate natural language requirements with code that is executed to generate tests. On the first iteration of the PyTcGen design, the intent was to use NLP tools such as CoreNLP to parse requirements and extract named entities such as input variable names. However, current NLP is geared towards more conversational and general language rather than the technology and industry-specific language used in formal requirements specifications. Additionally, requirements will

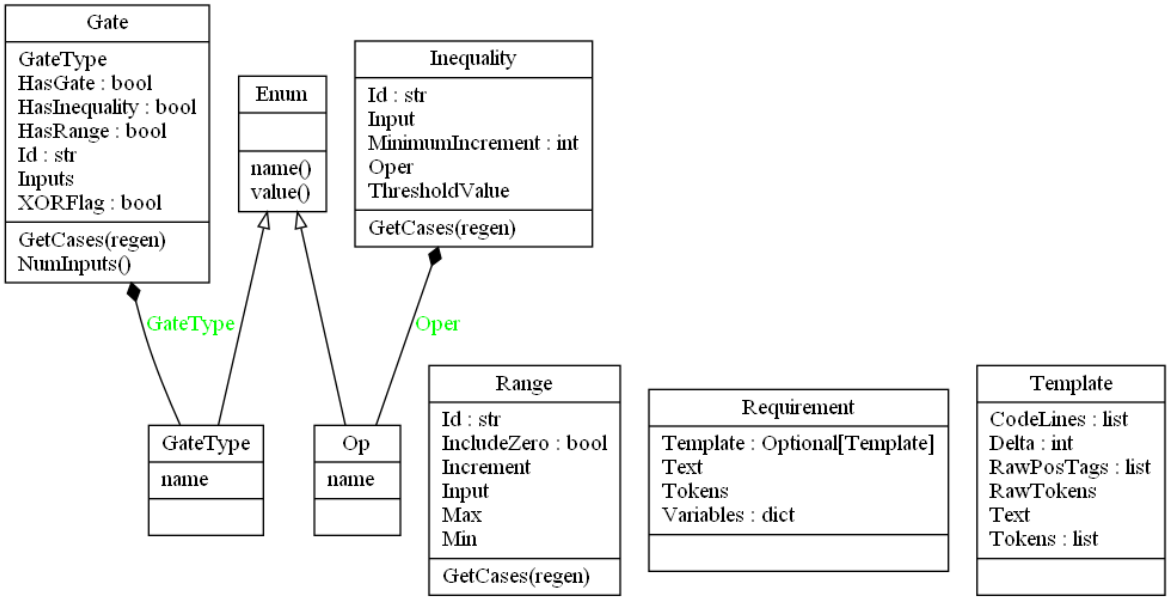

Figure 3.1: PyTcGen Class Diagram

frequently contain what could be called compound-named entities. That is, named entities that contain more than one word. For example, a requirement used during testing contained the term "Upper Desired Temperature." In this case, it represents an input value. However, both CoreNLP and the Python Natural Language Toolkit (NLTK) [7] cannot recognize this as a single entity. Other approaches also have had difficulties with NER from natural language requirements [22][37][43]. Based on the knowledge of state of the art in NER and the results from previous research by others, the decision to abstract over the NER issue was made, with the motivation being that the value of the work is not in advancing NER but in using NER to advance automatic test case generation. For these reasons, we developed the system for templating described here.

The templates proposed are natural language statements designed to match the requirement structure. A placeholder system is employed to indicate to the template matching algorithm (see § 3.2.1) that an entity is present. Template 3.1 is an example template used in testing the system. Placeholders are denoted with ${<name>}, where <name> is an alphanumeric value that is one-to-one with the named entities

within a given requirement. For example, in 3.1 the placeholder names {x1, x2, v1, v2} are used. The name x1 can be used in as many locations within the template as the template author chooses, provided the named entity in those places in a given matching requirement would also be the same value. Also, in this example, the naming pattern "xn" and "vn" was used. Where 'x' represents an input or output variable named entity, and 'v' represents a value named entity. But this nomenclature is our convention and not required.

"If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}." (3.1)

3.2.1 Template Matching

The parsed tokens from the requirements and templates are compared to find a match. If a match is found, the template and requirement are linked in memory, and execution continues. If no match is found in the set of templates, a measure of which of the available templates is most like the requirement is needed. Two problems would arise if PyTcGen compared requirements and templates in a simple token-to-token fashion. First, multi-word named entities would throw off the index-by-index comparison. Second, and more importantly, such an algorithm would make the progression in the comparison sequence the only measure of a template's "distance" from a requirement. That is, if comparing the sets of tokens r and t, then the distance δ is the measure of the closeness of t to r calculated as $\delta = n - i$, where n is the length of r and i is the first index where a mismatch between r and t was found. This means that the earlier a template failed to match a given requirement in a sequence, the greater the distance value calculated. This would create a problem if a template matched a requirement in all but the first token; the comparison calculation would give the template a greater distance value than a template that matched only the first two tokens. Thus, a solu-

tion that can handle non-synchronized comparison and gives an accurate measure of template-to-requirement distance regardless of where differences occur as needed.

The template comparison algorithm developed to overcome the above-mentioned challenges compares tokens in a given template against the requirement from the token array's start and end. Let c be the increasing index on t, m be the decreasing index on t, n the length of t, and n_r the length of r. With the initial value of $m = n - 1$ a template's δ from a requirement is calculated as:

$$\delta = \frac{c + (n_r - m)}{2} \tag{3.2}$$

where the values used in the calculation for c and m are the indices on t where the algorithm encountered a mismatch with r.

3.2.2 Template Associated Test Code

To illustrate, consider template 3.3 used in experimentation. This template corresponds to requirement 1.2. Each template is associated with Python code that utilizes the placeholder names from the template in conjunction with the classes in Template.py and TestGen.py to generate test cases.

> "When ${x1} is ${v1} and ${x2} is greater than or equal to ${v2} NMI then ${applicationName} shall set ${x3} to ${v3} and ${x4} to ${v4}."
> (3.3)

Now consider the associated code:

> "I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.GtEqTo, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())"
> (3.4)

In the associated code individual Python statements are separated with semicolons. Code statements are executed in a left-to-right fashion. For ease of reading, the individual code statements are as follows:

I1 = Inequality(None, x1, Op.EqualTo, v1) (3.5)

I2 = Inequality(None, x2, Op.GtEqTo, v2) (3.6)

gate = Gate("req1", GateType.AND, [I1, I2]) (3.7)

print(gate.GetCases()) (3.8)

Starting with the first statement 3.5, this code represents the "${x1} is ${v1}" part of 3.3. Statement 3.6 corresponds to "${x2} is greater than or equal to ${v2} NMI". Finally, the two inequalities are passed as the two inputs into an AND gate object in 3.7. The last statement 3.8 triggers the execution of the test generation for the requirement and prints the results. In this manner, a user can build complex gates with multiple inputs. By creating statements that establish inputs and input conditions that are internally passed to other gates and so on, a user can represent any Boolean logic expression.

3.3 Test Case Generation

The following section describes the algorithms used to generate tests for logic gates (and, or, and xor), inequalities, and ranges. The algorithms are designed to follow the $n + 1$ pattern as described in § 1.1.

3.3.1 Gates

3.3.1.1 AND

Test generation for AND gates is defined in Algorithm 1. For any AND gate with n-inputs, the requisite tests can be created by first listing all the values that make all inputs true, resulting in the output of the gate being true. Next, the remaining tests are generated by setting each input false, with the remaining inputs true with the gate's expectant result being false. To illustrate this, assume there is an AND gate with four inputs. The tests for this gate are listed in Table 3.1. In Table 3.1, 1 and 0 are used instead of True and False for space and clarity. Notice the progression from left to right of the '0' in the false tests. This is referred to as "walking the 0" when generating tests or "walking the 1" when generating other tests for other gate types.

3.3.1.2 OR & XOR

Test generation for OR and XOR gates is described in Algorithm 2. Generation is similar to AND gates as described in § 3.3.1.1 except that the initial test is all False/0 and the True/1 is walked through the inputs. Table 3.2 shows the tests required for a 4-input OR or XOR gate. Notice that the same set of test cases can test both gates. That is, recall from § 1.1 the $n+1$ test cases that demonstrate the independent effect of each input for an Or and XOR is the same set. This meets the letter of DO-178C, but it does not cover the case where an OR was implemented as an XOR in the code or

Table 3.1: 4-Input AND Gate

A	B	C	D	Result
1	1	1	1	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	0

Algorithm 1 AND Gate Test Generation

```
1: procedure AND(inputs)
2:     for i in inputs do                ▷ Add the positive path, or all true test case
3:         case[i] ← True
4:     end for
5:     case[result] ← True
6:     cases.append(case)
7:     for i in inputs do                ▷ Walk the 0 (False)
8:         for j in inputs do
9:             if i == j then
10:                case[i] ← False
11:            else
12:                case[j] ← True
13:            end if
14:        end for
15:        case[result] ← False
16:        cases.append(case)
17:    end for
18:    return cases
19: end procedure
```

Table 3.2: 4-Input OR or XOR Gate

A	B	C	D	Result
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	1	1

vice versa. PyTcGen gives the user an option to check for this type of error. When this option is exercised to test OR and XOR conditions are not implemented as the other, the number of test cases generated to cover a requirement is $n+2$ where the additional test is where all inputs are 1. These are shown in Tables 3.3 and 3.4. Providing this functionality is accomplished in the proposed solution by including the *checkXOR* flag in the method that generates tests for OR gates.

Algorithm 2 OR Gate Test Generation

```
1: procedure OR(inputs, checkXOR)
2:     for i in inputs do                          ▷ Add the all false test case
3:         case[i] ← False
4:     end for
5:     case[result] ← False
6:     cases.append(case)
7:     for i in inputs do                          ▷ Walk the 1 (True)
8:         for j in inputs do
9:             if i == j then
10:                case[i] ← True
11:            else
12:                case[j] ← False
13:            end if
14:        end for
15:        case[result] ← True
16:        cases.append(case)
17:    end for
18:    if checkXOR then
19:        for i in inputs do
20:            case[i] ← True
21:        end for
22:        case[result] ← True
23:        cases.append(case)
24:    end if
25:    return cases
26: end procedure
```

Table 3.3: 4-Input OR

A	B	C	D	Result
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	1	1
1	1	1	1	1

Table 3.4: 4-Input XOR

A	B	C	D	Result
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	1	1
1	1	1	1	0

3.3.2 Inequalities

The steps for generating tests for inequalities are described in Algorithm 3. As noted in § 1.1, tests for inequalities are based not only on the values on either side of the operator token but also on the properties of the data types and underlying hardware being used to develop the system under test. This is represented in Algorithm 3 with the *inc* parameter. This parameter represents the minimum increment on the input value possible given the system and data type used. For instance, if the data type employed was a float and the sensor providing the input had a resolution of 10^{-3} then the minimum increment would be 0.001. For each operation, the algorithm returns the requisite test values and the expected result of the condition. If the operation is =, the algorithm returns "equal to, *True*" and "not equal to, *False*." Refer to Algorithm 3.

Algorithm 3 Inequality Test Values Generation

```
1: procedure GenIneqVals(operation, threshold, inc)
2:     if operation is = then
3:         return [(Threshold, True), (threshold + inc, False)]
4:     else if operation is ≠ then
5:         return [(threshold + inc, True),(threshold, False)]
6:     else if operation is > then
7:         return [(threshold + inc, True),(threshold, False)]
8:     else if operation is >= then
9:         return [(threshold+inc, True),(threshold, True),(threshold−inc, False)]
10:    else if operation is < then
11:        return [(threshold − inc, True),(threshold, False)]
12:    else if operation is <= then
13:        return [(threshold−inc, True),(threshold, True),(threshold+inc, False)]
14:    end if
15: end procedure
```

3.3.3 Ranges

Tests for ranges, like inequalities (see § 3.3.2), are dependent on the properties of the variable that is used to store the value and the underlying system. Therefore, like Algorithm 3, Algorithm 4 includes the *inc* parameter for the same reasons. In addition to *inc*, Algorithm 4 also includes the *incldZero* parameter. Given the potential divide by 0 error that can occur, it is good practice to include 0 whenever a range encompasses it (see § 1.1). The *incldZero* flag allows users to include the value in generated tests optionally.

3.3.4 Complex Gates

Complex (or compound) gates are those gates whose inputs are themselves gates, inequalities, or a range check. The processing of complex gates is achieved similarly to the base objects described above. Algorithm 5 is an abridged version of that implemented in code. A full version of the algorithm is included in Appendix A. Gates are checked if they contain other gates, ranges, or inequalities as inputs. If they do not,

Algorithm 4 Range Test Values Generation

```
1: procedure GenRangeVals(min, max, inc, incldZero)
2:     falseCases.append(min − inc)
3:     trueCases.append(min)
4:     if incldZero & min ≠ 0 & max ≠ 0 then
5:         if min < 0 & 0 < max then
6:             trueCases.append(0)
7:         else
8:             falseCases.append(0)
9:         end if
10:    end if
11:    trueCases.append(max)
12:    falseCases.append(max + inc)
13:    return [trueCases, falseCases]
14: end procedure
```

then the input is a simple gate. If the gate contains other gates, GenerateGateCases is called recursively. Inequalities and Ranges cannot be complex, so the "GetCases()" method is called to force the generation of cases to build the complex gate cases. Once execution continues past these checks, the "root" gate's test cases are built by substituting input gate, range, or inequality cases for plain T/F results. For example, if an AND gate, 'A,' includes an AND gate, 'B,' as an input, then a case for A that requires a T result from B will include all cases with a true result from B's list of cases.

Algorithm 5 Generate Tests for Complex Gates

```
1: procedure GenerateGateCases(gate)
2:     if (¬gate.hasGate) and (¬gate.hasIneq) and (¬gate.hasRange) then
3:         if gate.Type is AND then
4:             gate.Cases = AndGate(gate.Inputs)
5:         else if gate.Type is OR then
6:             gate.Cases = OrGate(gate.Inputs)
7:         else
8:             gate.Cases = XorGate(gate.Inputs)
9:         end if
10:        return
11:    end if
12:    for i in gate.Inputs do
13:        if i is a Gate then
14:            GenerateGateCases(i)
15:        else if i is a Range or Inequality then
16:            i.GetCases()  ▷ This forces the cases to be generated on i for use below
17:        end if
18:    end for
19:    if gate.Type is AND then
20:        Link tests from input gates, ranges, Inequalities IAW AND algorithm
21:    else
22:        Link tests from input gates, ranges, Inequalities IAW OR/XOR algorithm
23:    end if
24: end procedure
```

Chapter 4

Experiments

Our experiments used twenty-five requirements to test and evaluate PyTcGen. Fifteen were reused from a previous study focused on interpreting natural language requirements [16]. One requirement was written from memory of safety-critical tests of aircraft software projects. The remaining eight requirements were generated from Requirement 4.3 as a test and example of the human-system co-evolutionary aspects of the proposed template system (see § 4.4).

4.1 Preliminary Experiments with Natural Language Processing

As was mentioned in Section 1.1, the initial intent for this thesis was to use Natural Language Processing (NLP) to extract Named Entities (NE) from requirements and generate test cases using the methods described. Where templates are employed now, PyTcGen would have used Named Entity Recognition (NER) to identify key terms. Preliminary experimentation with the Stanford CoreNLP yielded poor results for NER

on test requirements. Of particular trouble was identifying compound-named entities. However, while developing PyTcGen, a release was made that greatly improved the ability of CoreNLP to recognize multi-token named entities. Consider Requirement 4.1:

> "If the Regulator Mode equals NORMAL, the Output Regulator Status shall be set to On." (4.1)

For 4.1 CoreNLP produces the following for named entities:

```
[{
  ‘‘text’’: ‘‘the Regulator Mode’’,
  ‘‘type’’: ‘‘LAW’’,
  ‘‘start_char’’: 260,
  ‘‘end_char’’: 278
},{
  ‘‘text’’: ‘‘the Output Regulator Status’’,
  ‘‘type’’: ‘‘LAW’’,
  ‘‘start_char’’: 294,
  ‘‘end_char’’: 321
}]
```

In this example, CoreNLP does a good job identifying both the input and output named entities. However, the values of each, "NORMAL" and "On", are not recognized as entities. Notice that the classification given to each is "LAW", which does not make sense for this application. Lastly, CoreNLP does not recognize "equals" as a comparator in the manner the requirement intends. The missing tagging and classification make sense in the context of the standard language forms that CoreNLP is designed to work with. An NLP toolkit that can recognize not only multi-token named entities such as these but also "engineering speak" is what is needed to replace manual template generation. See § 5.1.1 for additional details.

Also used in NER experimentation was the Python-based Natural Language Toolkit (NLTK). The results with NLTK were similar to earlier results with CoreNLP. Multi-token named entities are not recognized. Output for Requirement 4.1 using the NLTK was as follows:

```
(S
  If/IN
  the/DT
  Regulator/NNP
  Mode/NNP
  equals/NNS
  (ORGANIZATION NORMAL/NNP)
  ,/,
  the/DT
  (ORGANIZATION Output/NNP)
  Regulator/NNP
  Status/NNP
  shall/MD
  be/VB
  set/VBN
  to/TO
  On/IN
  ./.)
```

In this example, NLTK fails to identify "Regulator Mode" but does identify "NORMAL." While the multi-token named entity "Output Regulator Status" is partially identified with "Output" being labeled an organization.

4.2 Template Matching

To evaluate the effectiveness of the template matching algorithm, the requirements and accompanying templates were varied to test PyTcGen's ability to match different requirements and template structures. After initial development efforts, the algorithm matched all 25 requirements under test, and introducing inaccuracies in templates

caused matching failures. The current template matching code is case-sensitive and cannot compensate for grammatical or semantic variances. An out-of-place comma or miss-capitalization will result in a failed match. As an example of failed template matching, consider the below example:

```
I wasn't able to find a template that matched the following requirement:
If the Regulator Interface Failure is set to True or the Regulator
Internal Failure is set to True or the Status attribute of the Current
Temperature is not set to Valid, the Regulator Status shall be set to
False.
This is the template I found that I think is the closest:
If the ${x1} is set to ${v1}, or the ${x2} is set to ${v2} or the ${a1}
attribute of the ${c1} is not set to ${v3}, the ${x4} shall be set
to ${v4}.
```

The reason for the failed match is apparent when one notices the comma following ${v1}.

4.3 Test Case Generation

Tests were developed by hand by the author for each requirement to evaluate the effectiveness of the test generation algorithm. We compared the automatically generated tests against those generated by the algorithm. In all cases, the algorithm generated the correct set of test cases. Consider Requirement 4.2:

> "If the Regulator Mode equals INIT, and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL." (4.2)

Requirement 4.2 can be represented with Diagram 4.1.

Requirement 4.2 is a simple 2-input AND gate. To cover this requirement test engineer would be expected to generate three test cases. Written as a table, those tests would be:

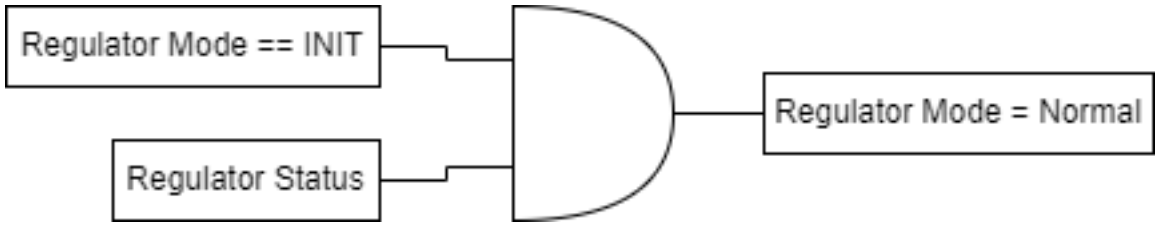

Figure 4.1: Example Requirement

Table 4.1: Example Requirement Test Cases

Regulator Mode	Regulator Status	Result
INIT	True	True
¬ INIT	True	False
INIT	False	False

When Requirement 4.2 is processed by PyTcGen the following output is produced:

```
[{'ineq_zLdJMqWFwRwQ': [('Regulator Mode', 'INIT', True)],
  'ineq_UdyeEENGBWzU': [('Regulator Status', True, True)], 'Result': True},
 {'ineq_zLdJMqWFwRwQ': [('Regulator Mode', 'not INIT', False)],
  'ineq_UdyeEENGBWzU': [('Regulator Status', True, True)], 'Result': False},
 {'ineq_zLdJMqWFwRwQ': [('Regulator Mode', 'INIT', True)],
  'ineq_UdyeEENGBWzU': [('Regulator Status', 2, False)], 'Result': False}]
```

The first values "ineq_..." are unique identifiers generated by the code. Unique identifiers are generated automatically when identifiers are not provided. In the test values produced, the reader can see that the requisite T, F, F tests are generated with the same input values as specified in the manually generated Table 4.1. In all examples tested, PyTcGen produced the expected list of tests.

4.4 Co-evolution

Previous solutions have either tried to understand natural language requirements fully [44], or force the user into writing in a subset of English [17]. Neither of these

approaches works 100% of the time and can require even experienced users to update their capabilities. As discussed in § 1.3.1, co-evolution is a side-effect in existing approaches. In this section we discuss how co-evolution is the intent and feature of the proposed solution.

A vital aspect of the proposed solution is the co-evolution offered by the development requirements and templates in tandem or asynchronously. Users could develop requirements and a matching template simultaneously if one does not already exist. Or, templates could be created before requirements, anticipating the requirement style and forms to be developed. One development scenario is for an engineering firm to adopt the tool on a trial project. Then after the successful trial, users would expand its use to other projects. The corpus of templates is expanded, tailored, and refined with each additional use.

But along with the expansion of available templates comes feedback to requirement authors in the form of feedback cycles. I.e., a requirement is written and run through PyTcGen, but a matching template is not found. Instead, the tool presents the closest match found. The author can add a new template, but they can also alter the requirement to match the existing template if it is close. This is a different way to arrive at a controlled subset of English to use for requirements specifications. Still, because it is developed as part of the requirements development and not *a priori*, the difficulties in expressions should be significantly less.

The more beneficial path to be following in this approach is enforced requirement complexity reduction. The tool, as currently written, does not support multi-line complex requirements as inputs nor similarly complex templates. While PyTcGen could support additional complexity in later versions, enforced simplicity is advantageous. Consider Requirement 4.3. This requirement is taken from an actual safety-critical project. It is used with permission, provided the input and output (I/O) names were

changed to obfuscate the industry and company. The structure of the requirement is unchanged. We changed each occurrence of I/O and value names in a one-to-one fashion. The device/system name was changed to "The Widget." The device/system connection became "Widget Connection." Name structure was also preserved such that multi-word terms that used underscores are still the same structure. Even as challenging to understand as it is, Requirement 4.3 can still be represented as a Boolean expression and, therefore, as a Boolean logic diagram as in 4.2. At the time 4.3 was borrowed, the correctness of 4.2 was confirmed by the systems engineer responsible for writing the requirement.

(4.3)

The Widget **shall** set Widget Connection to indicate "Widget Does Not Have Widget Connection" when the following conditions occur on a widget with Widget Connection:

- The Widget has received less than two Valid Recorder messages with the EWidget_Status field set to "Active", or any Valid Recorder messages with the EWidget_Status field set to "Inactive", for the duration of DB_EWidget_Status_Validation_Time
  within DB_EWidget_Status_Timeout of exiting the least recently Exited Square being reported as Occupied in the
  Blinker_Square_Occupancy_Message due to the Widget not being able to reconfirm Widget Connection since exiting the Square, and
- DB_EWidget_Status_Timeout has expired since the least recently Exited Square was reported as Occupied in the
  Blinker_Square_Occupancy_Message due to the Widget not being able to reconfirm Widget Connection since exiting the Square, and
- The Widget is reporting DB_Intel_Square_Count or more Exited Squares as Occupied, and
- The Active Widget Type has
  DB_EWidget_Required_For_Widget_Integrity set to "Enabled" and
  DB_EWidget_Comm_Method set to "WidgetSchmidget".

Requirement 4.3 is a complicated requirement for a few reasons. First, using complex variable names with mixed formats instead of a single standardized naming struc-

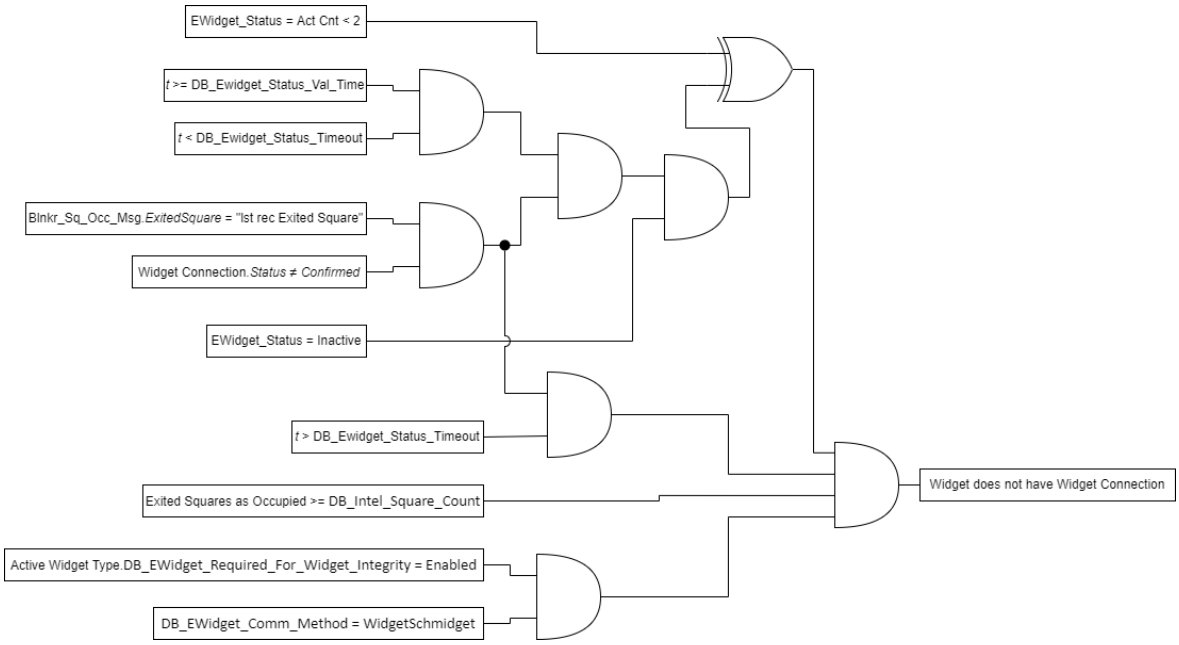

Figure 4.2: Example Real-World Complex Requirement

ture can make tracking inputs, outputs, and expected values challenging. Second, the requirement contains multiple uses of the same variable names. Lastly, as shown in Figure 4.2, there are eight gates defined by this single requirement in only four bullets.

As an example of the co-evolutionary power of the proposed solution, simplifying Requirement 4.3 to fit the input constraints of PyTcGen also provides answers to the difficulties described above. To decompose this requirement into more straightforward requirements, the individual gates were isolated and intermediary values were introduced. The intermediary values were given the names InterVal$n$, where n is an arbitrary number assignment. Consider the portion of Figure 4.2 in Figure 4.3. The output from this gate is not external and, therefore, can be relabeled without affecting any system interface. Rewriting the corresponding portion of Requirement 4.3, this gate's parent requirement becomes:

(4.4)

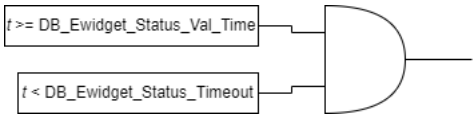

Figure 4.3: Intermediate Value 1 Gate

> When Time is greater than or equal to DB_Ewidget_Status_Val_Time and less than DB_Ewidget_Status_Timeout than InterVal1 shall be True.

Continuing in a similar manner, Requirement 4.3 can be rewritten as the set of requirements:

(4.5) When Time is greater than or equal to DB_Ewidget_Status_Val_Time and less than DB_EWidget_Status_Timeout than InterVal1 shall be True.

(4.6) If the Exited Square in the Blinker_Square_Occupancy_Message is equal to the least recently Exited Square being reported as Occupied due to Widget not being able to Confirm Widget Connection then InterVal2 shall be True.

(4.7) If InterVal1 and InterVal2 are True than InterVal3 shall be True.

(4.8) If InterVal3 is True and EWidget_Status is Inactive than InterVal4 shall be True.

(4.9) If the count of Valid Recorder messages with EWidget_Status equal to Active is less than 2 or InterVal4 is True than InterVal5 shall be True.

(4.10) If InterVal2 is True and Time is greater than DB_EWidget_Status_Timeout than InterVal6 shall be True.

(4.11) If the Active Widget Type has DB_EWidget_Required_For_Widget_Integrity set to Enabled and DB_EWidget_Comm_Method set to WidgetSchmidget than InterVal7 shall be True.

(4.12) If InterVal5, InterVal6, and InterVal7 are True and the Exited Squares as Occupied count is greater than DB_Intel_Square_Count than the Widget shall set Widget Connection to "Widget Does Not Have Widget Connection."

It is arguable that this set of requirements is easier to understand and therefore implement and test successfully. If this were a real in-context engineering effort, then

meaningful values could also be used in the place of the InterVal$n$ nomenclature, further enhancing the comprehensibility. These requirements were derived after the initial sixteen requirements, which means we had already developed templates. This is analogous to new requirements being added to an existing PyTcGen-style system. Predictably, initial runs with the requirements did not match any of these requirements to templates. For example, running Requirement 4.8 yielded:

```
I wasn't able to find a template that matched the following requirement:
If InterVal3 is True and EWidget_Status is Inactive than InterVal4
shall be True.
This is the template I found that I think is the closest:
If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, and the ${a1}
attribute of the ${c1} is set to ${v3}, the ${x3} shall be set to ${v4}.
```

This approximation makes sense given the operation of the template matching algorithm. The left→right scan would have found only *'If'* matching, but the right→left scan would have been able to reach *InerVal4*. Looking at the template found and comparing it to the requirement in question, there is an extra structure in the template accounting for an attribute of a class. (See § 5.1.2) Attempting to match this template Requirement 4.8 becomes:

(4.13)

> If the InterVal3 is set to True, and the EWidget_Status is set to Inactive, the InterVal4 shall be set to True.

In this form, the requirement did match an existing template. Similarly, Requirements 4.7 and 4.10 can be rewritten as:

(4.14) If the InterVal1 is set to True, and the InterVal2 is set to True, the InterVal3 shall be set to True.

(4.15) If the InterVal2 equals True, and the Time is greater than the DB_EWidget_Status_Timeout, the InterVal6 shall be set to True.

Finally, after creating templates and code for the remaining five requirements, PyTcGen successfully generated test cases for all eight requirements derived from Requirement 4.3.

Chapter 5

Conclusion

This thesis presents three challenges facing safety-critical software development and testing. First, Safety-critical test requires an understanding of Boolean logic expressions, equivalence classes, and range and boundary conditions. Second, test development is a time-consuming and costly process that is necessarily so to ensure all tests are identified. Third, frequent changes to requirements drives the frequent rederivation of associated tests. The proposed solution, PyTcGen, successfully mitigates these problems by programmatically generating all requisite tests from input requirements. The proposed solution has the added benefit of creating a co-evolutionary relationship between the system and requirement authors that reduces requirement ambiguity while increasing the applicability of the tool to additional projects through broadening the corpus of identifiable requirement forms.

Experimentation with PyTcGen showed that the template approach is a viable means for test case generation. By utilizing requirements that are representative of real-world safety-critical requirements and successfully generating the requisite tests for those requirements, we have demonstrated PyTcGen's suitability for use. Further, the simplification of a complicated requirement taken directly from industry demonstrates

the co-evolutionary power of the template approach in driving requirement authors to requirement forms that are far less ambiguous.

As discussed in § 1.3.1 and demonstrated in § 4.4 co-evolution is a necessity of any solution to the problem of automated test generation. While a feedback loop exists as a part of other proposed tools, co-evolution a designed feature of the solution we presented in this thesis. The template matching feature provides an immediate recommendation as to requirement alterations. Users can alter the requirement or the template, or they can add additional templates. Over time this process both expands the number of available templates and drives requirement uniformity.

5.1 Future Research

There are several future research and development areas where PyTcGen could be improved.

5.1.1 Named Entity Recognition

The current solution relies on requirement templates to identify key terms and Boolean logic in requirements. The template system was necessary because NER in NLP solutions could not reliably detect both multi-token named entities and I/O value literals. The Stanford CoreNLP could detect some multi-token named entities, and the Python NLTK was capable of recognizing some literals. Still, neither could recognize all terms in the requirement set used, indicating that template-based systems will be necessary. Combining outputs from both tools was also insufficient for generating tests. What is needed is a NER model trained on a large engineering documentation corpus. The major obstacle to any such effort will be the collection of such a corpus. Engineering organizations are reticent to release, for public research, the information they

consider proprietary. If researchers could overcome this, and a NER model was developed that correctly identifies engineering terms as inputs, outputs, states, Booleans, literals, etc. In that case, we could quickly adapt the proposed solution to utilize such a tool in the place of the current template system.

5.1.2 Template Matching

There are several opportunities for improvements in the proposed template-based solution. The current solution is sensitive to minor differences between the requirements and templates. It is possible to adapt the system to make variances in capitalization and punctuation irrelevant. Employing comparison tables or other data structures that allow for equivalent terms would increase template matching flexibility considerably. Phrases such as "x is set to y" and "x equals y" could be matched with the same template if "is set to" and "equals" were known by the system to be equivalent. Term equivalency would drastically reduce the number of required templates.

We can improve PyTcGen by adapting the solution to handle multi-line requirements and templates. Multi-line requirements would allow for more complex syntactical structures. Making configurable, the number of close templates returned when a matching template is not found would improve the ability of the user to change requirements or add or update templates. Increasing the number of templates displayed if a requirement is not matched to a template may improve the user's ability to make an informed decision. Currently, only the template with the highest δ value is displayed.

Recall from equation 3.2 in § 3.2.1 that the right-to-left template comparison utilizes the value n_r. Using this parameter times the calculation to the length of the requirement. The value δ measures how much a given template matches the requirement. In code, there is another parameter n that could be used. n is the length of a given template in the number of tokens. Utilizing n in 3.2 ties δ to the length of

the template. Developmental testing utilizing n in place of n_r tended to favor shorter templates. Calculating based on the length of the requirement and template together may produce better template suggestions in situations where small portions of the requirement are matched.

Lastly, the template matching algorithm employs both a left-to-right and a right-to-left comparison. This approach effectively ensures that templates that have differences at either end are not outright rejected. Longer templates that may match requirements in middle portions will receive low δ values. A third parsing option that is likely to address this is to start in the middle of a template and requirement and traverse both left and right. Combining all three parsing methods should mean that all similarities between any given requirement and template are found.

Bibliography

[1] Imran Ahsan, Wasi Haider Butt, Mudassar Adeel Ahmed, and Muhammad Waseem Anwar. A comprehensive investigation of natural language processing techniques and tools to generate automated test cases. *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*, pages 1–10, 2017.

[2] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Ničkovic, Rupert Schlick, Didier Simoneau, and Stefan Tiran. Integration of Requirements Engineering and Test-Case Generation Via OSLC. *2014 14th International Conference on Quality Software*, pages 117–126, 2014.

[3] Sai Chaithra Allala, Juan P. Sotomayor, Dionny Santiago, Tariq M. King, and Peter J. Clarke. Towards Transforming User Requirements to Test Cases Using MDE and NLP. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 2:350–355, 2019.

[4] Andres Arellano, Edward Zontek-Carney, and Mark A. Austin. Frameworks for Natural Language Processing of Textual Requirements. *International Journal On Advances in Systems and Measurements*, pages 230–240, 2015.

[5] José L. Barros-Justo, Fabiane B.V. Benitti, and Ania L. Cravero-Leal. Software patterns and requirements engineering activities in real-world settings: A systematic mapping study. *Computer Standards & Interfaces*, 58:23–42, 2018.

[6] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli. Verification of Quasi-Synchronous Systems with Uppaal. *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pages 8A4–1–8A4–12, 2014.

[7] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[8] Eman S Btoush and Mustafa M Hammad. Generating ER Diagrams from Requirement Specifications Based On Natural Language Processing. *International Journal of Database Theory and Application*, 8(2):61–70, 2015.

[9] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Test case generation from natural language requirements based on SCR specifications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1217–1222, 2013.

[10] Ram Chatterjee and Kalpana Johari. A prolific approach for automated generation of test cases from informal requirements. *ACM SIGSOFT Software Engineering Notes*, 35(5):1–11, 2010.

[11] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software engineering journal*, 9(5):193–200, 1994.

[12] National Center for Biomedical Computing. i2b2: Informatics for Integrating Biology & the Bedside.

[13] A Cooper. Equivalence Classes, 11 2018.

[14] Themistoklis Diamantopoulos, Michael Roth, Andreas Symeonidis, and Ewan Klein. Software requirements as an application domain for natural language processing. *Language Resources and Evaluation*, 51(2):495–524, 2017.

[15] Rongrong Fu, Xiaohong Bao, and Tingdi Zhao. Generic Safety Requirements Description Templates for the Embedded Software. In *017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*, pages 1477–1481, 5 2017.

[16] Shalini Ghosh, Natarajan Shankar, Patrick Lincoln, Daniel Elenius, Wenchao Li, and Wilfrid Steiener. Automatic Requirements Specification Extraction from Natural Language (ARSENAL). Technical report, 2014.

[17] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Automated formalization of structured natural language requirements. *Information and Software Technology*, 137:106590, 2021.

[18] Axel Hoffmann, Matthias Söllner, Holger Hoffmann, and Jan Marco Leimeister. Towards trust-based software requirement patterns. In *2012 Second IEEE International Workshop on Requirements Patterns (RePa)*, pages 7–11, 2012.

[19] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (tte) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 22–33, 2005.

[20] Taciana Novo Kudo, Renato F. Bulcão Neto, and Auri M. R. Vincenzi. A conceptual metamodel to bridging requirement patterns to test patterns. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, page 155–160, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Dilek Küçük. Joint Named Entity Recognition and Stance Detection in Tweets. *arXiv*, 2017.

[22] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural Architectures for Named Entity Recognition. *Proceedings of NAACL*, 2016.

[23] Mathias Landhauber and Walter F Tichy. Automated Test-Case Generation by Cloning. *2012 7th International Workshop on Automation of Software Test (AST)*, 1:83–88, 2012.

[24] Ji Young Lee, Franck Dernoncourt, and Peter Szolovits. Transfer Learning for Named-Entity Recognition with Neural Networks. *arXiv*, 2017.

[25] David L. Lempia and Steven P. Miller. Requirements Engineering Management Handbook. Technical report, 2009.

[26] Pan Liu, Yihao Li, and Huaikou Miao. Transition algebra for software testing. *IEEE Transactions on Reliability*, 70(4):1438–1454, 2021.

[27] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[28] Abha Moitra, Kit Siu, Andrew W. Crapo, Michael Durling, Meng Li, Panagiotis Manolios, Michael Meiners, and Craig McMillan. Automating requirements analysis and test case generation. *Requirements Engineering*, 24(3):341–364, 2019.

[29] Afrah Naeem, Zeeshan Aslam, and Munam Ali Shah. Analyzing Quality of Software Requirements; A Comparison Study on NLP Tools. In *Proceedings of the 25th International Conference on Automation & Computing*, pages 1–6, Lancaster University, Lancaster UK, 9 2019.

[30] C. Nebut, S. Pickin, Y. Le Traon, and J.-M. Jezequel. Automated requirements-based generation of test cases for product families. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 263–266, 2003.

[31] Oyindamola Olajubu. Automated test case generation from domain-specific high-level requirement models. 2018.

[32] Oyindamola Olajubu, Suraj Ajit, Mark Johnson, Scott Turner, Scott Thomson, and Mark Edwards. Automated test case generation from domain specific models of high-level requirements. *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pages 505–508, 2015.

[33] Oyindamola Olajubu, Suraj Ajit, Mark Johnson, Scott Turner, Scott Thomson, and Mark Edwards. Automated Test Case Generation from High-Level Logic Requirements Using Model Transformation Techniques. *2017 9th Computer Science and Electronic Engineering (CEEC)*, pages 178–182, 2017.

[34] Laboratory for Computational Physiology. MIMIC.

[35] Jesús Poza, Valentín Moreno, Anabel Fraga, and José María Álvarez Rodríguez. Genetic Algorithms: A Practical Approach to Generate Textual Patterns for Requirements Authoring. *Applied Sciences*, 11(23):11378, 2021.

[36] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178C), 12 2011.

[37] Yanyao Shen, Hyokun Yun, Zachary C Lipton, Yakov Kronrod, and Animashree Anandkumar. Deep Active Learning for Named Entity Recognition. *arXiv*, 2017.

[38] Rajvir Singh, Anita Singhrova, and Rajesh Bhatia. Test Case Generation Tools - A Review. *International Journal of Electronics Engineering*, 10(2):586–596, 11 2018.

[39] Robert Skeel. Roundoff Error and the Patriot Missile. *SIAM News*, 25(4):11, July 1992.

[40] Alvaro Veizaga, Mauricio Alferez, Damiano Torre, Mehrdad Sabetzadeh, and Lionel Briand. On systematically building a controlled natural language for functional requirements. *Empirical Software Engineering*, 26(4):79, 2021.

[41] Ravi Prakash Verma and Rizwan Beg. Generation of Test Cases from Software Requirements Using Natural Language Processing. *2013 6th International Conference on Emerging Trends in Engineering and Technology*, pages 140–147, 2013.

[42] Syazwani Yahya, Massila Kamalrudin, Safiah Sidek, Munaliza Jaimun, Junaidah Yusof, Ang Kean Hua, and Paran Gani. A Review Paper: Security Requirement Patterns for a Secure Software Development. *2019 1st International Conference on Artificial Intelligence and Data Sciences (AiDAS)*, 00:146–151, 2019.

[43] Jie Yang, Yue Zhang, and Fei Dong. Neural Reranking for Named Entity Recognition. *arXiv*, 2017.

[44] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol-Valeriu Chioasca, and Riza T. Batista-Navarro. Natural Language Processing for Requirements Engineering. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021.

Appendix A

Complex Gate Algorithm

Algorithm 6 Generate Tests for Complex Gates (Complete)

```
1: procedure GenerateGateCases(gate)
2:     if (¬gate.hasGate) and (¬gate.hasIneq) and (¬gate.hasRange) then
3:         if gate.Type is AND then
4:             gate.Cases = AndGate(gate.Inputs)
5:         else if gate.Type is OR then
6:             gate.Cases = OrGate(gate.Inputs)
7:         else
8:             gate.Cases = XorGate(gate.Inputs)
9:         end if
10:        return
11:    end if
12:    for i in gate.Inputs do
13:        if i is a Gate then
14:            GenerateGateCases(i)
15:        else if i is a Range or Inequality then
16:            i.GetCases()  ▷ This forces the cases to be generated on i for use below
17:        end if
18:    end for
19:    cases as List
20:    case as Dictionary
21:    if gate.Type is AND then
22:        See algorithm 7
23:    else
24:        See algorithm 8
25:    end if
26:    return cases
27: end procedure
```

Algorithm 7 Complex Gate Subpart 1

```
1: for i in gate.Inputs do                                   ▷ Positive Path/Base case
2:     if i is a Gate, Inequality, or Range then
3:         case[i] = All cases for i where result is True
4:     else
5:         case[i] = True
6:     end if
7: end for
8: case[result] = True
9: cases.append(case)
10: for i in gate.Inputs do                                  ▷ Walk the 0
11:     for j in gate.Inputs do
12:         if i == j then
13:             if i is a Gate, Inequality, or Range then
14:                 case[i] = All cases for i where result is False
15:             else
16:                 case[i] = False
17:             end if
18:         else
19:             if i is a Gate, Inequality, or Range then
20:                 case[i] = All cases for i where result is True
21:             else
22:                 case[i] = True
23:             end if
24:         end if
25:     end for
26:     case[result] = False
27:     cases.append(case)
28: end for
```

Algorithm 8 Complex Gate Subpart 2

```
1: for i in gate.Inputs do                                   ▷ Base case (All False)
2:     if i is a Gate, Inequality, or Range then
3:         case[i] = All cases for i where result is False
4:     else
5:         case[i] = False
6:     end if
7: end for
8: case[result] = False
9: cases.append(case)
10: for i in gate.Inputs do                                  ▷ Walk the 1
11:     for j in gate.Inputs do
12:         if i == j then
13:             if i is a Gate, Inequality, or Range then
14:                 case[i] = All cases for i where result is True
15:             else
16:                 case[i] = True
17:             end if
18:         else
19:             if i is a Gate, Inequality, or Range then
20:                 case[i] = All cases for i where result is False
21:             else
22:                 case[i] = False
23:             end if
24:         end if
25:     end for
26:     case[result] = True
27:     cases.append(case)
28: end for
29: if CheckXOR or gate.Type is XOR then
30:     for i in gate.Inputs do
31:         if i is a Gate, Inequality, or Range then
32:             case[i] = All cases for i where result is True
33:         else
34:             case[i] = True
35:         end if
36:     end for
37:     case[result] = (gate.Type == OR)
38:     cases.append(case)
39: end if
```

Appendix B

Requirements

1. When AirGroundStatus is OnGround and GroundSpeed is greater than or equal to 120 NMI then TheSystem shall set AirGroundStatus to InAir and AirGroundStatusOverride to True.

2. If the Regulator Mode equals INIT, the Output Regulator Status shall be set to Init.

3. If the Regulator Mode equals NORMAL, the Output Regulator Status shall be set to On.

4. If the Regulator Mode equals FAILED, the Output Regulator Status shall be set to Failed.

5. If the Status attribute of the Lower Desired Temperature or the Upper Desired Temperature equals Invalid, the Regulator Interface Failure shall be set to True.

6. If the Status attribute of the Lower Desired Temperature and the Upper Desired Temperature equals Valid, the Regulator Interface Failure shall be set to False.

7. If the Regulator Mode equals INIT, the Heat Control shall be set to Off.

8. If the Regulator Mode equals NORMAL, and the Current Temperature is less than the Lower Desired Temperature, the Heat Control shall be set to control On.

9. If the Regulator Mode equals NORMAL, and the Current Temperature is greater than the Upper Desired Temperature, the Heat Control shall be set to Control Off.

10. If the Regulator Mode equals FAILED, the Heat Control shall be set to Control Off.

11. If the Regulator Interface Failure is set to False, and the Regulator Internal Failure is set to False, and the Status attribute of the Current Temperature is set to Valid, the Regulator Status shall be set to True.

12. If the Regulator Interface Failure is set to True, or the Regulator Internal Failure is set to True or the Status attribute of the Current Temperature is not set to Valid, the Regulator Status shall be set to False.

13. The Regulator Mode shall be initialized to INIT.

14. If the Regulator Mode equals INIT, and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.

15. If the Regulator Mode is set to NORMAL, and the Regulator Status is set to False, the Regulator Mode shall be set to FAILED.

16. If the Regulator Mode is set to INIT, and the Regulator Init Timeout is set to True, the Regulator Mode shall be set to FAILED.

17. When Time is greater than or equal to DB_Ewidget_Status_Val_Time and less than DB_EWidget_Status_Timeout than InterVal1 shall be True.

18. If the Exited Square in the Blinker_Square_Occupancy_Message is equal to the least recently Exited Square being reported as Occupied due to Widget not being able to Confirm Widget Connection then InterVal2 shall be True.

19. If the InterVal1 is set to True, and the InterVal2 is set to True, the InterVal3 shall be set to True.

20. If the InterVal3 is set to True, and the EWidget_Status is set to Inactive, the InterVal4 shall be set to True.

21. If the count of Valid Recorder messages with EWidget_Status equal to Active is less than 2 or InterVal4 is True than InterVal5 shall be True.

22. If the InterVal2 equals True, and the Time is greater than the DB_EWidget_Status_Timeout, the InterVal6 shall be set to True.

23. If the Active Widget Type has DB_EWidget_Required_For_Widget_Integrity set to Enabled and DB_EWidget_Comm_Method set to WidgetSchmidget than InterVal7 shall be True.

24. If InterVal5, InterVal6, and InterVal7 are True and the Exited Squares as Occupied count is greater than DB_Intel_Square_Count than the Widget shall set Widget Connection to “Widget Does Not Have Widget Connection”.

Appendix C

Templates

1. When ${x1} is ${v1} and ${x2} is greater than or equal to ${v2} NMI then ${appName} shall set ${x3} to ${v3} and ${x4} to ${v4}.
2. If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
3. The ${x1} shall be initialized to ${v1}.
4. If the ${x1} equals ${v1}, and the ${x2} equals ${v2}, the ${x3} shall be set to ${v3}.
5. If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, the ${x3} shall be set to ${v3}.
6. If the ${a1} attribute of the ${c1} or the ${c2} equals ${v1}, the ${x1} shall be set to ${v2}.
7. If the ${a1} attribute of the ${c1} and the ${c2} equals ${v1}, the ${x1} shall be set to ${v2}.
8. If the ${x1} equals ${v1}, and the ${x2} is less than the ${v2}, the ${x3} shall be set to ${v3}.
9. If the ${x1} equals ${v1}, and the ${x2} is greater than the ${v2}, the ${x3} shall be set to ${v3}.
10. If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, and the ${a1} attribute of the ${c1} is set to ${v3}, the ${x3} shall be set to ${v4}.
11. If the ${x1} is set to ${v1}, or the ${x2} is set to ${v2} or the ${a1} attribute of the ${c1} is not set to ${v3}, the ${x4} shall be set to ${v4}.
12. When ${x1} is greater than or equal to ${v1} and less than ${v2} than ${x2} shall be ${v3}.

13. If the ${x1} in the ${c1} is equal to the ${x2} then ${x3} shall be ${v1}.

14. If the ${x1} is less than ${v1} or ${x2} is ${v2} than ${x3} shall be ${v3}.

15. If the ${c1} has ${x1} set to ${v1} and ${x2} set to ${v2} than ${x3} shall be ${v3}.

16. If ${x1}, ${x2}, and ${x3} are ${v1} and the ${x4} is greater than ${v2} than the Widget shall set ${x5} to "${v3}".

Appendix D

Template Code

1. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.GtEqTo, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())
2. I1 = Inequality(None, x1, Op.EqualTo, v1);print(I1.GetCases())
3. I1 = Inequality(None, x1, Op.EqualTo, v1);print(I1.GetCases())
4. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.EqualTo, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())
5. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.EqualTo, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())
6. I1 = Inequality(None, a1 + "." + c1, Op.EqualTo, v1);I2 = Inequality(None, a1 + "." + c2, Op.EqualTo, v1);gate = Gate("req1", GateType.OR, [I1, I2]); print(gate.GetCases())
7. I1 = Inequality(None, a1 + "." + c1, Op.EqualTo, v1);I2 = Inequality(None, a1 + "." + c2, Op.EqualTo, v1);gate = Gate("req1", GateType.AND, [I1, I2]); print(gate.GetCases())
8. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.LessThan, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())
9. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.GrtrThan, v2);gate = Gate("req1", GateType.AND, [I1, I2]);print(gate.GetCases())
10. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.EqualTo, v2);I3=Inequality(None, a1 + "." + c1, Op.EqualTo, v3);gate = Gate("req1", GateType.AND, [I1, I2, I3]);print(gate.GetCases())

11. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.EqualTo, v2);I3=Inequality(None, a1 + "." + c1, Op.NtEqTo, v3);gate = Gate("req1", GateType.OR, [I1, I2, I3]);print(gate.GetCases())

12. I1 = Inequality(None, x1, Op.GtEqTo, v1);I2 = Inequality(None, x1, Op.LessThan, v2);gate = Gate("req2", GateType.AND, [I1, I2]);print(gate.GetCases())

13. I1 = Inequality(None, c1 + "." + x1, Op.EqualTo, x2);print(I1.GetCases())

14. I1 = Inequality(None, x1, Op.LessThan, v1);I2 = Inequality(None, x2, Op.EqualTo, v2);gate = Gate("reqPi", GateType.OR, [I1, I2], True);print(gate.GetCases())

15. I1 = Inequality(None, c1 + "." + x1, Op.EqualTo, v1);I2 = Inequality(None, c1 + "." + x2, Op.EqualTo, v2);gate = Gate("req42", GateType.AND, [I1, I2]);print(gate.GetCases())

16. I1 = Inequality(None, x1, Op.EqualTo, v1);I2 = Inequality(None, x2, Op.EqualTo, v1);I3 = Inequality(None, x3, Op.EqualTo, v1);I4 = Inequality(None, x4, Op.GrtrThan, v2);gate = Gate("Bob", GateType.AND, [I1, I2, I3, I4]);print(gate.GetCases())

Appendix E

Sample Output

The following is the output from PyTcGen when run utilizing the requirements, templates, and code specified in the previous appendices.

```
**************************************************************************
Requirement: When AirGroundStatus is OnGround and GroundSpeed is greater
    than or equal to 120 NMI then TheSystem shall set AirGroundStatus to
    InAir and AirGroundStatusOverride to True.
Template: When ${x1} is ${v1} and ${x2} is greater than or equal to ${v2}
    NMI then ${appName} shall set ${x3} to ${v3} and ${x4} to ${v4}.
Test cases:
[{'ineq_mOwDJlbPpyVs': [('AirGroundStatus', 'OnGround', True)], '
    ineq_FrijsTqtiIVy': [('GroundSpeed', 121, True)], 'Result': True}, {'
    ineq_mOwDJlbPpyVs': [('AirGroundStatus', 'not OnGround', False)], '
    ineq_FrijsTqtiIVy': [('GroundSpeed', 121, True)], 'Result': False}, {'
    ineq_mOwDJlbPpyVs': [('AirGroundStatus', 'OnGround', True)], '
    ineq_FrijsTqtiIVy': [('GroundSpeed', 119, False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Mode equals INIT, the Output Regulator Status
    shall be set to Init.
Template: If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
Test cases:
[('Regulator Mode', 'INIT', True), ('Regulator Mode', 'not INIT', False)]
**************************************************************************
Requirement: If the Regulator Mode equals NORMAL, the Output Regulator
    Status shall be set to On.
Template: If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
Test cases:
[('Regulator Mode', 'NORMAL', True), ('Regulator Mode', 'not NORMAL', False)
    ]
**************************************************************************
```

```
Requirement: If the Regulator Mode equals FAILED, the Output Regulator
   Status shall be set to Failed.
Template: If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
Test cases:
[('Regulator Mode', 'FAILED', True), ('Regulator Mode', 'not FAILED', False)
   ]
**********************************************************************
Requirement: If the Status attribute of the Lower Desired Temperature or the
    Upper Desired Temperature equals Invalid, the Regulator Interface
   Failure shall be set to True.
Template: If the ${a1} attribute of the ${c1} or the ${c2} equals ${v1}, the
    ${x1} shall be set to ${v2}.
Test cases:
[{'ineq_GslwrSfewKLV': [('Status.Lower Desired Temperature', 'not Invalid',
   False)], 'ineq_vWWgwiTuMAuj': [('Status.Upper Desired Temperature', 'not
   Invalid', False)], 'Result': False}, {'ineq_GslwrSfewKLV': [('Status.
   Lower Desired Temperature', 'Invalid', True)], 'ineq_vWWgwiTuMAuj': [('
   Status.Lower Desired Temperature', 'not Invalid', False)], 'Result': True
   }, {'ineq_GslwrSfewKLV': [('Status.Upper Desired Temperature', 'not
   Invalid', False)], 'ineq_vWWgwiTuMAuj': [('Status.Upper Desired
   Temperature', 'Invalid', True)], 'Result': True}]
**********************************************************************
Requirement: If the Status attribute of the Lower Desired Temperature and
   the Upper Desired Temperature equals Valid, the Regulator Interface
   Failure shall be set to False.
Template: If the ${a1} attribute of the ${c1} and the ${c2} equals ${v1},
   the ${x1} shall be set to ${v2}.
Test cases:
[{'ineq_GrUtNceAGeLO': [('Status.Lower Desired Temperature', 'Valid', True)
   ], 'ineq_tLnPIKIMRHWj': [('Status.Upper Desired Temperature', 'Valid',
   True)], 'Result': True}, {'ineq_GrUtNceAGeLO': [('Status.Lower Desired
   Temperature', 'not Valid', False)], 'ineq_tLnPIKIMRHWj': [('Status.Upper
   Desired Temperature', 'Valid', True)], 'Result': False}, {'
   ineq_GrUtNceAGeLO': [('Status.Lower Desired Temperature', 'Valid', True)
   ], 'ineq_tLnPIKIMRHWj': [('Status.Upper Desired Temperature', 'not Valid
   ', False)], 'Result': False}]
**********************************************************************
Requirement: If the Regulator Mode equals INIT, the Heat Control shall be
   set to Off.
Template: If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
Test cases:
[('Regulator Mode', 'INIT', True), ('Regulator Mode', 'not INIT', False)]
**********************************************************************
Requirement: If the Regulator Mode equals NORMAL, and the Current
   Temperature is less than the Lower Desired Temperature, the Heat Control
```

```
    shall be set to control On.
Template: If the ${x1} equals ${v1}, and the ${x2} is less than the ${v2},
    the ${x3} shall be set to ${v3}.
Test cases:
[{'ineq_bEgaKkmyBUwE': [('Regulator Mode', 'NORMAL', True)], '
    ineq_AaXYWZCHFNAS': [('Current Temperature', 'is less than Lower Desired
    Temperature', True)], 'Result': True}, {'ineq_bEgaKkmyBUwE': [('Regulator
     Mode', 'not NORMAL', False)], 'ineq_AaXYWZCHFNAS': [('Current
    Temperature', 'is less than Lower Desired Temperature', True)], 'Result':
     False}, {'ineq_bEgaKkmyBUwE': [('Regulator Mode', 'NORMAL', True)], '
    ineq_AaXYWZCHFNAS': [('Current Temperature', 'is equal to Lower Desired
    Temperature', False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Mode equals NORMAL, and the Current
    Temperature is greater than the Upper Desired Temperature, the Heat
    Control shall be set to Control Off.
Template: If the ${x1} equals ${v1}, and the ${x2} is greater than the ${v2
    }, the ${x3} shall be set to ${v3}.
Test cases:
[{'ineq_zILzzEcOYtad': [('Regulator Mode', 'NORMAL', True)], '
    ineq_LOiIjgCnxvLP': [('Current Temperature', 'is greater than Upper
    Desired Temperature', True)], 'Result': True}, {'ineq_zILzzEcOYtad': [('
    Regulator Mode', 'not NORMAL', False)], 'ineq_LOiIjgCnxvLP': [('Current
    Temperature', 'is greater than Upper Desired Temperature', True)], '
    Result': False}, {'ineq_zILzzEcOYtad': [('Regulator Mode', 'NORMAL', True
    )], 'ineq_LOiIjgCnxvLP': [('Current Temperature', 'is equal to Upper
    Desired Temperature', False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Mode equals FAILED, the Heat Control shall be
    set to Control Off.
Template: If the ${x1} equals ${v1}, the ${x2} shall be set to ${v2}.
Test cases:
[('Regulator Mode', 'FAILED', True), ('Regulator Mode', 'not FAILED', False)
    ]
**************************************************************************
Requirement: If the Regulator Interface Failure is set to False, and the
    Regulator Internal Failure is set to False, and the Status attribute of
    the Current Temperature is set to Valid, the Regulator Status shall be
    set to True.
Template: If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, and
    the ${a1} attribute of the ${c1} is set to ${v3}, the ${x3} shall be set
    to ${v4}.
Test cases:
[{'ineq_gPWuGPsAYpvL': [('Regulator Interface Failure', False, True)], '
    ineq_rrzdnhBanSlX': [('Regulator Internal Failure', False, True)], '
```

```
    ineq_fUzSRiHvyKcn': [('Status.Current Temperature', 'Valid', True)], '
    Result': True}, {'ineq_gPWuGPsAYpvL': [('Regulator Interface Failure', 1,
False)], 'ineq_rrzdnhBanSlX': [('Regulator Internal Failure', False, True)],
     'ineq_fUzSRiHvyKcn': [('Status.Current Temperature', 'Valid', True)], '
    Result': False}, {'ineq_gPWuGPsAYpvL': [('Regulator Interface Failure',
    False, True)], 'ineq_rrzdnhBanSlX': [('Regulator Internal Failure', 1,
    False)], 'ineq_fUzSRiHvyKcn': [('Status.Current Temperature', 'Valid',
    True)], 'Result': False}, {'ineq_gPWuGPsAYpvL': [('Regulator Interface
    Failure', False, True)], 'ineq_rrzdnhBanSlX': [('Regulator Internal
    Failure', False, True)], 'ineq_fUzSRiHvyKcn': [('Status.Current
    Temperature', 'not Valid', False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Interface Failure is set to True, or the
    Regulator Internal Failure is set to True or the Status attribute of the
    Current Temperature is not set to Valid, the Regulator Status shall be
    set to False.
Template: If the ${x1} is set to ${v1}, or the ${x2} is set to ${v2} or the
    ${a1} attribute of the ${c1} is not set to ${v3}, the ${x4} shall be set
    to ${v4}.
Test cases:
[{'ineq_sDsGZQevDRIM': [('Regulator Interface Failure', 2, False)], '
    ineq_MHpRtevMaOQY': [('Regulator Internal Failure', 2, False)], '
    ineq_rcapqaVAbNLx': [('Status.Current Temperature', 'Valid', False)], '
    Result': False}, {'ineq_sDsGZQevDRIM': [('Regulator Interface Failure',
    True, True)], 'ineq_MHpRtevMaOQY': [('Regulator Interface Failure', 2,
    False)], 'ineq_rcapqaVAbNLx': [('Regulator Interface Failure', 2, False)
    ], 'Result': True}, {'ineq_sDsGZQevDRIM': [('Regulator Internal Failure',
     2, False)], 'ineq_MHpRtevMaOQY': [('Regulator Internal Failure', True,
    True)], 'ineq_rcapqaVAbNLx': [('Regulator Internal Failure', 2, False)],
    'Result': True}, {'ineq_sDsGZQevDRIM': [('Status.Current Temperature', '
    Valid', False)], 'ineq_MHpRtevMaOQY': [('Status.Current Temperature', '
    Valid', False)], 'ineq_rcapqaVAbNLx': [('Status.Current Temperature', '
    not Valid', True)], 'Result': True}]
**************************************************************************
Requirement: The Regulator Mode shall be initialized to INIT.
Template: The ${x1} shall be initialized to ${v1}.
Test cases:
[('Regulator Mode', 'INIT', True), ('Regulator Mode', 'not INIT', False)]
**************************************************************************
Requirement: If the Regulator Mode equals INIT, and the Regulator Status
    equals True, the Regulator Mode shall be set to NORMAL.
Template: If the ${x1} equals ${v1}, and the ${x2} equals ${v2}, the ${x3}
    shall be set to ${v3}.
Test cases:
```

```
[{'ineq_jbOcefRhOpxI': [('Regulator Mode', 'INIT', True)], '
    ineq_LfooTjvCrban': [('Regulator Status', True, True)], 'Result': True},
    {'ineq_jbOcefRhOpxI': [('Regulator Mode', 'not INIT', False)], '
    ineq_LfooTjvCrban': [('Regulator Status', True, True)], 'Result': False},
     {'ineq_jbOcefRhOpxI': [('Regulator Mode', 'INIT', True)], '
    ineq_LfooTjvCrban': [('Regulator Status', 2, False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Mode is set to NORMAL, and the Regulator
    Status is set to False, the Regulator Mode shall be set to FAILED.
Template: If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, the $
    {x3} shall be set to ${v3}.
Test cases:
[{'ineq_HpqEZTsmwHeE': [('Regulator Mode', 'NORMAL', True)], '
    ineq_LqCAAWKBmztQ': [('Regulator Status', False, True)], 'Result': True},
     {'ineq_HpqEZTsmwHeE': [('Regulator Mode', 'not NORMAL', False)], '
    ineq_LqCAAWKBmztQ': [('Regulator Status', False, True)], 'Result': False
    }, {'ineq_HpqEZTsmwHeE': [('Regulator Mode', 'NORMAL', True)], '
    ineq_LqCAAWKBmztQ': [('Regulator Status', 1, False)], 'Result': False}]
**************************************************************************
Requirement: If the Regulator Mode is set to INIT, and the Regulator Init
    Timeout is set to True, the Regulator Mode shall be set to FAILED.
Template: If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, the $
    {x3} shall be set to ${v3}.
Test cases:
[{'ineq_qxkJPdklfCdp': [('Regulator Mode', 'INIT', True)], '
    ineq_JrJpEoObQJOC': [('Regulator Init Timeout', True, True)], 'Result':
    True}, {'ineq_qxkJPdklfCdp': [('Regulator Mode', 'not INIT', False)], '
    ineq_JrJpEoObQJOC': [('Regulator Init Timeout', True, True)], 'Result':
    False}, {'ineq_qxkJPdklfCdp': [('Regulator Mode', 'INIT', True)], '
    ineq_JrJpEoObQJOC': [('Regulator Init Timeout', 2, False)], 'Result':
    False}]
**************************************************************************
Requirement: When Time is greater than or equal to
    DB_Ewidget_Status_Val_Time and less than DB_EWidget_Status_Timeout than
    InterVal1 shall be True.
Template: When ${x1} is greater than or equal to ${v1} and less than ${v2}
    than ${x2} shall be ${v3}.
Test cases:
[{'ineq_rwUdSxwaUGGS': [('Time', 'is equal to DB_Ewidget _Status_Val_Time',
    True)], 'ineq_iuFpvPJUQIas': [('Time', 'is less than DB_EWidget_Status
    _Timeout', True)], 'Result': True}, {'ineq_rwUdSxwaUGGS': [('Time', 'is
    less than DB_Ewidget _Status_Val_Time', False)], 'ineq_iuFpvPJUQIas': [('
    Time', 'is less than DB_EWidget_Status _Timeout', True)], 'Result': False
    }, {'ineq_rwUdSxwaUGGS': [('Time', 'is greater than DB_Ewidget
    _Status_Val_Time', True)], 'ineq_iuFpvPJUQIas': [('Time', 'is equal to
```

```
   DB_EWidget_Status _Timeout', False)], 'Result': False}]
**********
Requirement: If the Exited Square in the Blinker_Square_Occupancy_Message is
     equal to the least recently Exited Square being reported as Occupied due
     to Widget not being able to Confirm Widget Connection then InterVal2
    shall be True.
Template: If the ${x1} in the ${c1} is equal to the ${x2} then ${x3} shall
    be ${v1}.
Test cases:
[('Blinker_Square_Occupancy _Message.Exited Square', 'least recently Exited
    Square being reported as Occupied due to Widget not being able to Confirm
     Widget Connection', True), ('Blinker_Square_Occupancy _Message.Exited
    Square', 'not least recently Exited Square being reported as Occupied due
     to Widget not being able to Confirm Widget Connection', False)]
**********
Requirement: If the InterVal1 is set to True, and the InterVal2 is set to
    True, the InterVal3 shall be set to True.
Template: If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, the $
    {x3} shall be set to ${v3}.
Test cases:
[{'ineq_qBjmBrrdAtZS': [('InterVal1', True, True)], 'ineq_gJMDzMRumIWZ': [('
    InterVal2', True, True)], 'Result': True}, {'ineq_qBjmBrrdAtZS': [('
    InterVal1', 2, False)], 'ineq_gJMDzMRumIWZ': [('InterVal2', True, True)],
     'Result': False}, {'ineq_qBjmBrrdAtZS': [('InterVal1', True, True)], '
    ineq_gJMDzMRumIWZ': [('InterVal2', 2, False)], 'Result': False}]
**********
Requirement: If the InterVal3 is set to True, and the EWidget_Status is set
    to Inactive, the InterVal4 shall be set to True.
Template: If the ${x1} is set to ${v1}, and the ${x2} is set to ${v2}, the $
    {x3} shall be set to ${v3}.
Test cases:
[{'ineq_niSUafxfEiPM': [('InterVal3', True, True)], 'ineq_TzNWdBlWEIEP': [('
    EWidget_Status', 'Inactive', True)], 'Result': True}, {'ineq_niSUafxfEiPM
    ': [('InterVal3', 2, False)], 'ineq_TzNWdBlWEIEP': [('EWidget_Status', '
    Inactive', True)], 'Result': False}, {'ineq_niSUafxfEiPM': [('InterVal3',
     True, True)], 'ineq_TzNWdBlWEIEP': [('EWidget_Status', 'not Inactive',
    False)], 'Result': False}]
**********
Requirement: If the count of Valid Recorder messages with EWidget_Status
    equal to Active is less than 2 or InterVal4 is True than InterVal5 shall
    be True.
Template: If the ${x1} is less than ${v1} or ${x2} is ${v2} than ${x3} shall
     be ${v3}.
Test cases:
```

```
[{'ineq_rLMcSADRwkXO': [('count of Valid Recorder messages with
    EWidget_Status equal to Active', 2, False)], 'ineq_egQuHbXHHmej': [('
    InterVal4', 2, False)], 'Result': False}, {'ineq_rLMcSADRwkXO': [('count
    of Valid Recorder messages with EWidget_Status equal to Active', 1, True)
    ], 'ineq_egQuHbXHHmej': [('count of Valid Recorder messages with
    EWidget_Status equal to Active', 2, False)], 'Result': True}, {'
    ineq_rLMcSADRwkXO': [('InterVal4', 2, False)], 'ineq_egQuHbXHHmej': [('
    InterVal4', True, True)], 'Result': True}, {'ineq_rLMcSADRwkXO': [('count
     of Valid Recorder messages with EWidget_Status equal to Active', 1, True
    )], 'ineq_egQuHbXHHmej': [('InterVal4', True, True)], 'Result': True}]
**********
Requirement: If the InterVal2 equals True, and the Time is greater than the
    DB_EWidget_Status_Timeout, the InterVal6 shall be set to True.
Template: If the ${x1} equals ${v1}, and the ${x2} is greater than the ${v2
    }, the ${x3} shall be set to ${v3}.
Test cases:
[{'ineq_YycKDmyhBNYB': [('InterVal2', True, True)], 'ineq_uktgGVQetEnZ': [('
    Time', 'is greater than DB_EWidget_Status _Timeout', True)], 'Result':
    True}, {'ineq_YycKDmyhBNYB': [('InterVal2', 2, False)], '
    ineq_uktgGVQetEnZ': [('Time', 'is greater than DB_EWidget_Status _Timeout
    ', True)], 'Result': False}, {'ineq_YycKDmyhBNYB': [('InterVal2', True,
    True)], 'ineq_uktgGVQetEnZ': [('Time', 'is equal to DB_EWidget_Status
    _Timeout', False)], 'Result': False}]
**********
Requirement: If the Active Widget Type has
    DB_EWidget_Required_For_Widget_Integrity set to Enabled and
    DB_EWidget_Comm_Method set to WidgetSchmidget than InterVal7 shall be
    True.
Template: If the ${c1} has ${x1} set to ${v1} and ${x2} set to ${v2} than ${
    x3} shall be ${v3}.
Test cases:
[{'ineq_cjtjtJcBbIdd': [('Active Widget Type.DB_EWidget_Required _For_Widget
     _Integrity', 'Enabled', True)], 'ineq_UgCyLXGRWHwg': [('Active Widget
    Type.DB_EWidget _Comm_Method', 'WidgetSchmidget', True)], 'Result': True
    }, {'ineq_cjtjtJcBbIdd': [('Active Widget Type.DB_EWidget_Required
    _For_Widget _Integrity', 'not Enabled', False)], 'ineq_UgCyLXGRWHwg': [('
    Active Widget Type.DB_EWidget _Comm_Method', 'WidgetSchmidget', True)], '
    Result': False}, {'ineq_cjtjtJcBbIdd': [('Active Widget Type.
    DB_EWidget_Required _For_Widget _Integrity', 'Enabled', True)], '
    ineq_UgCyLXGRWHwg': [('Active Widget Type.DB_EWidget _Comm_Method', 'not
    WidgetSchmidget', False)], 'Result': False}]
**********
Requirement: If InterVal5, InterVal6, and InterVal7 are True and the Exited
    Squares as Occupied count is greater than DB_Intel_Square_Count than the
    Widget shall set Widget Connection to ‘‘Widget Does Not Have Widget
```

```
   Connection’’.
Template: If ${x1}, ${x2}, and ${x3} are ${v1} and the ${x4} is greater than
    ${v2} than the Widget shall set ${x5} to ‘‘${v3}’’.
Test cases:
[{’ineq_QeJZpZpWRWgp’: [(’InterVal5’, True, True)], ’ineq_zukwodMCxQRc’: [(’
   InterVal6’, True, True)], ’ineq_XllQFuGpuemi’: [(’InterVal7’, True, True)
   ], ’ineq_hPCIwecOyHmM’: [(’Exited Squares as Occupied count’, ’is greater
    than DB_Intel_Square _Count’, True)], ’Result’: True}, {’
   ineq_QeJZpZpWRWgp’: [(’InterVal5’, 2, False)], ’ineq_zukwodMCxQRc’: [(’
   InterVal6’, True, True)], ’ineq_XllQFuGpuemi’: [(’InterVal7’, True, True)
   ], ’ineq_hPCIwecOyHmM’: [(’Exited Squares as Occupied count’, ’is greater
    than DB_Intel_Square _Count’, True)], ’Result’: False}, {’
   ineq_QeJZpZpWRWgp’: [(’InterVal5’, True, True)], ’ineq_zukwodMCxQRc’: [(’
   InterVal6’, 2, False)], ’ineq_XllQFuGpuemi’: [(’InterVal7’, True, True)],
    ’ineq_hPCIwecOyHmM’: [(’Exited Squares as Occupied count’, ’is greater
   than DB_Intel_Square _Count’, True)], ’Result’: False}, {’
   ineq_QeJZpZpWRWgp’:
[(’InterVal5’, True, True)], ’ineq_zukwodMCxQRc’: [(’InterVal6’, True, True)
   ], ’ineq_XllQFuGpuemi’: [(’InterVal7’, 2, False)], ’ineq_hPCIwecOyHmM’:
   [(’Exited Squares as Occupied count’, ’is greater than DB_Intel_Square
   _Count’, True)], ’Result’: False}, {’ineq_QeJZpZpWRWgp’: [(’InterVal5’,
   True, True)], ’ineq_zukwodMCxQRc’: [(’InterVal6’, True, True)], ’
   ineq_XllQFuGpuemi’: [(’InterVal7’, True, True)], ’ineq_hPCIwecOyHmM’: [(’
   Exited Squares as Occupied count’, ’is equal to DB_Intel_Square _Count’,
   False)], ’Result’: False}]
```