Detecting Inconsistency in Functional Software Requirements

Gboyega Sanni
Iowa State University

John K. Jackman
Iowa State University, jkj@iastate.edu

Follow this and additional works at: https://lib.dr.iastate.edu/imse_conf
Part of the Operational Research Commons

Recommended Citation
https://lib.dr.iastate.edu/imse_conf/169
Detecting Inconsistency in Functional Software Requirements

Abstract
Success in software development depends on the availability of complete, consistent, and unambiguous functional software requirements. Inconsistencies in software requirements can propagate problems throughout the development cycle. We introduce the concept of a quantitative measure for detecting inconsistencies, namely, Potential Structural Inconsistency (PSI). This measure is derived from a structural model for a given set of requirements. We show how this measure can be determined using a case study with known inconsistencies.

Keywords
inconsistency, detection, software and requirements

Disciplines
Operational Research

Comments
Detecting Inconsistency in Functional Software Requirements
Gboyega Sanni and John Jackman
Industrial and Manufacturing Systems Engineering, Iowa State University, Ames, Iowa, US.

Abstract
Success in software development depends on the availability of complete, consistent, and unambiguous functional software requirements. Inconsistencies in software requirements can propagate problems throughout the development cycle. We introduce the concept of a quantitative measure for detecting inconsistencies, namely, Potential Structural Inconsistency (PSI). This measure is derived from a structural model for a given set of requirements. We show how this measure can be determined using a case study with known inconsistencies.

Keywords: inconsistency, detection, software and requirements

1 INTRODUCTION
Software is an essential asset for information intensive industries such as, finance, aviation, manufacturing, and retail. Government, military, corporate and customer stakeholders invest significant resources in software systems in order to increase profit, automate business processes, and improve competitiveness. Before a software system is developed, stakeholders define a set of functional software requirements. This set of requirements describes the functions that a customer needs in order to achieve their objectives. The process of defining functional requirements can lead to inconsistencies between requirements in the set. An inconsistency can be viewed as two or more requirements that disagree with each other. Early detection and removal of inconsistencies can help reduce costs in downstream activities of the Software Development Life Cycle (SDLC). For example, the European Space Agency’s Ariane 5 rocket exploded [9] during launch, resulting in a loss more than $7 billion. The explosion was attributed to an attempt in the software to convert a floating-point number into a 16-bit integer without checking for overflow. Though it has not been proven that inconsistencies in functional requirements caused this incident, it is certainly plausible that inconsistent functional requirements can lead to inconsistencies in software modules.

One approach for detecting inconsistencies is to model software requirements as goals and use goal monitors to detect inconsistencies [14]. A limitation with this approach is that software requirements must be specified as goals, whereas most requirements are defined in natural language. Hunter [10] proposed a novel approach of detecting inconsistencies using formal logic. Proof theorems were used to determine whether requirements are logically inconsistent. Zowghi et al [20] translated software requirements defined in natural language into propositional logic. The propositional logic is then validated with a theorem prover.

Our premise is that there are different types of inconsistencies and that their detection is associated with the type of representation used for the software requirements though overlaps may exist. If this is true, then our expectation is to find logical inconsistencies using Hunter or Zowghi’s methods. We propose a method that examines the structural consistency of a set of requirements. In our method, functional software requirements are transformed into a structural representation, which is then analyzed for any structural inconsistencies. We introduce the concept of potential, partial or relative inconsistency in this context. In the next section, we show how the requirements can be modeled from a structural perspective. This is followed by a description of the methods used to examine consistency in the model. Finally, we conclude with a well-known example from the literature and discuss the implications of our method in detecting inconsistencies.

2 MODEL FORMULATION
The term, inconsistent, has been defined as “If a reason, idea, reason, idea, opinion, etc. is inconsistent, different parts of it do not agree, or it does not agree with something else” [1]. Suppose that software requirements described with natural language are represented as a set of statements (written in English) that describe the functions of the software. Zowghi [20] has shown that it is possible to transform a set of statements in English into a corresponding set of logical statements. Logical inconsistencies are detected using an automated theorem prover ([4]; [11]). Using a similar approach, we transform a set of English statement into a set of corresponding structural models. A structural inconsistency occurs if different elements of the structural representation do not agree with each other. A disagreement is defined as the existence of dissimilarity between two or more structural elements. Potential
structural inconsistency (PSI) is specified as the degree to which the elements of two or more structures disagree, hence the use of the term “potential.”

Some overlap in elements must exist before inconsistency can be detected in a set of requirements ([16]; [17]; [12]). Overlap is a measure of the similarity between two structures. For two identical structures, we can assign a value of 1 for similarity, since there is complete overlap from wholly shared or identical elements. When the structures are completely different, the similarity is 0 since there is no overlap. If we alter elements in one of two identical structures, then we introduce a potential structural inconsistency (PSI). We need a measure of structural similarity before a PSI can be detected. The degree of inconsistency indicates relative, partial or potential structural inconsistencies.

3 METHODOLOGY
3.1 TRANSFORMATION OF FUNCTIONAL SOFTWARE REQUIREMENTS INTO STRUCTURAL MODELS

We use an Entity Relationship Model (ERM) as the basis of our structural model. Previous attempts to transform text in natural language into ERMs used parsers to identify the Parts Of Speech (POS) and then mapped the POS to elements in the final model ([2]; [18]). One of the difficulties is that parsers tend to perform poorly when identifying the POS. Brill [3] showed that a rule-based tagger could be used to obtain the POS tags of words in statements with the context (i.e., word sequence) still preserved.

Using a similar approach as Brill, we can generate a structural model given a sequence of POS in a set of requirement statements. The ERM described by Chen [5] consists of cardinalities, entities, attributes and relationships. Chen [6] proposed that each member of the ERM could be mapped directly to English sentences. Though Chen’s recommendations are simplistic, they can serve as a starting point for generating the model. Validation of the model is necessary to show that the original requirements have been accurately represented.

3.2 SET THEORY REPRESENTATION OF ENTITY RELATIONSHIP MODEL

Let \( E = \{ e_1, e_2, e_3, \ldots, e_n \} \) be the set of \( n \) entities in a model. Entity \( i \) has an attribute set, \( A_i = \{ a_{i1}, a_{i2}, a_{i3}, \ldots, a_{im} \} \), describing the entity. The relationships between entities are defined by a set of Boolean variables, \( R = \{ r_{i1}, r_{i2}, r_{i3}, \ldots, r_{im} \} \), indicating whether a relationship exists. The cardinality of the relationships is a corresponding set of tuples, \( C = \{ c_{i1}, c_{i2}, c_{i3}, \ldots, c_{im} \} \), that show the number of instances that are involved of each entity type and \( c_{ij} = (\nu_i, \nu_j) \), where, \( \nu_i \) is the number of instances for entity \( i \). An ERM, \( \Gamma \), is defined as a three-tuple given by

\[
\Gamma = (E, R, C).
\]  

(1)

3.3 DETECTION OF POTENTIAL STRUCTURAL INCONSISTENCY IN STRUCTURAL MODELS

Based on the definitions of inconsistencies related to disagreement and similarity, the detection of PSI is indicated in part by the measure of similarity between two ERMs. Two measures of similarity were considered based on Tversky’s measure of similarity [19] and Edit Distances [13]. We used Tversky’s measure due to its simple set-based representation. Rodriguez [15] used Tversky’s measure of similarity in finding semantic similarity among entity classes. The measure uses a similarity function for determining similar cardinalities, entities, attributes and relationships at different levels. Tversky’s measure of similarity is used as a metric in the determination of PSI in functional software requirements.

Let \( X \) and \( Y \) be defined as two ERMs according to (1). We can perform set operations on elements of \( X \) and \( Y \) to determine which elements the models have in common and which ones are different. Let \( \Phi_K(X \ Y) \) be all elements of type \( K \) (e.g., entities) and \( \Phi_K(X \ Y) \) be the set of elements of type \( K \) common to both models. We can define those elements in \( X \) but not in \( Y \) as \( \Phi_K(X - Y) = \Phi_K(X \ Y) - \Phi_K(Y) \). Similarly for \( Y \) we obtain \( \Phi_K(Y - X) = \Phi_K(X \ Y) - \Phi_K(X) \). Using the elements from sets \( E, R, \) and \( A \), we can obtain composite cardinality functions based on equal weights for the three sets giving us

\[
F(X \ Y) = |\Phi_E(X \ Y)| + |\Phi_K(X \ Y)| + |\Phi_A(X \ Y)|,
\]
\[ F(X - Y) = \Phi_E (X - Y) + \Phi_R (X - Y) + \Phi_A (X - Y), \text{ and} \]
\[ F(Y - X) = \Phi_E (Y - X) + \Phi_R (Y - X) + \Phi_A (Y - X). \]

We can then define the similarity of \( X \) and \( Y \) as the difference between the number of common elements and those elements that are not in common, giving us
\[ S(X, Y) = \gamma_1 F(X \cap Y) - \gamma_2 F(X - Y) - \gamma_3 F(Y - X) \]

where, \( \gamma_1, \gamma_2 \) and \( \gamma_3 \geq 0 \), are assignable weights assumed to be \( \gamma_1=\gamma_2=\gamma_3 =1 \). The weights are used to assign importance to each function depending on the context of similarity. To normalize \( S(X, Y) \) between 0 and 1, it is necessary to determine the upper and lower bounds of \( S(X, Y) \). The upper bound corresponds to \( \min(|X|,|Y|) \) which would be the largest number of elements that \( X \) and \( Y \) could have in common. Likewise, the lower bound corresponds to \( - (|X| + |Y|) \) representing the condition when all elements are different. The normalized similarity (i.e., PSI) is given by
\[ \Psi(X, Y) = \frac{S(X, Y) - (- (|X| + |Y|))}{\min(|X|,|Y|) - (- (|X| + |Y|))} = \frac{S(X, Y) + (|X| + |Y|)}{\min(|X|,|Y|) + (|X| + |Y|)}. \]

4 CASE STUDY
4.1 OVERVIEW

The London Ambulance Service (LAS) [7] has been used as a case study in requirements engineering. When the LAS system was deployed on October 26, 1992, many problems occurred. For example, a subsystem (Automatic Vehicle Locating System (AVLS)) failed to track the location and status of some of the dispatch units [8]. The system allocated multiple vehicles to the same incidents.

A record series of fatal incidents lead to the termination of the LAS dispatch system. One fatal incident was due to the late arrival of an ambulance. Fatalities were attributed to the LAS and the heavy volume of calls and messages that besieged the system. The report shows that some areas were not fully defined in the Software Requirements Specifications (SRS). The requirements used by Zowghi [20] are shown in Table 1:

<table>
<thead>
<tr>
<th>Requirements ID</th>
<th>4.1.1 Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRC.1</td>
<td>A medical emergency is either an illness or an accident</td>
</tr>
<tr>
<td>IRC.2</td>
<td>When an operator receives a phone call concerning a medical emergency, the operator should dispatch the nearest available ambulance</td>
</tr>
<tr>
<td>IRC.3</td>
<td>When an operator receives a phone call concerning a non-medical emergency, the operator should not dispatch an ambulance and he should transfer the phone call to another service.</td>
</tr>
<tr>
<td>OM.1</td>
<td>When an operator receives a phone call, the operator should dispatch the nearest available ambulance.</td>
</tr>
<tr>
<td>OM.2</td>
<td>When an operator receives a phone call, if an ambulance is not the nearest available, then the operator should not dispatch that ambulance.</td>
</tr>
</tbody>
</table>
4.2 **ANALYSIS OF THE CASE STUDY**
Using our set notation, we can define \( \Gamma \) for IRC.2 as shown in Tables 2, 3, and 4.

<table>
<thead>
<tr>
<th>Table 2: IRC.2 Relationship Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>call</strong></td>
</tr>
<tr>
<td><strong>emergency</strong></td>
</tr>
<tr>
<td><strong>ambulance</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: IRC.2 Cardinality Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>call</strong></td>
</tr>
<tr>
<td><strong>emergency</strong></td>
</tr>
<tr>
<td><strong>ambulance</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4: IRC.2 Attribute Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
</tr>
</tbody>
</table>

After performing the same operation for OM.1, we can find \( \Psi(X, Y) \) for the pair IRC.2 and OM.1 as follows.

\[
\Phi_E(X \ U Y) = \{ \text{operator, call, ambulance} \}
\]

\[
\Phi_B(X \ U Y) = \{ \text{receives, should dispatch} \}
\]

\[
\Phi_E(X \ U Y) = \{ \text{phone, nearest, available} \}
\]

\[
F(X, Y) = 3 + 2 + 3 = 8
\]

\[
F(X - Y) = 3
\]

\[
F(Y - X) = 0
\]

\[
S(X, Y) = 8 - 3 - 0 = 5
\]

\[
\min(|X||Y|) = |Y| = 8
\]

\[
|X| + |Y| = 11 + 8 = 19
\]

\[
\Psi(X, Y) = \frac{5 + 19}{8 + 19} = .89
\]

PSI was determined for each pairing of requirements as shown in Table 5.
The magnitude of PSI suggests the order in which the requirements should be examined and may also imply importance. The IRC.2 and OM.1 pair yields the highest PSI value, suggesting that this pair of requirements should be reexamined. On closer examination of the two requirements, OM.1 could be interpreted as an ambulance being dispatched whenever a phone call is received (e.g., wrong number or information request). Another interesting observation is that those pairs with IRC.1 consistently have the lowest values. Therefore, this implies that IRC.1 is the requirement with the least PSI.

4.3 CONCLUSION
The potential for structural inconsistency in functional software requirements presents new opportunities in requirements engineering, inconsistency detection, conceptual modeling, and metrics in reasoning about PSI. It confirms that numerous forms of representation can be used to reason about different types of inconsistency. Our initial results indicate that PSI could be used to prioritize requirements. Closely related to the concept of PSI is the possible reexamination or revision of requirement statements. PSI can provide the basis for a more sophisticated requirement analysis model that can be used in software engineering practice.

References