RESOLVING CONFLICTING STAKEHOLDER VALUE PROPOSITIONS
IN VALUE-BASED REQUIREMENT ENGINEERING

A Project

Presented to the faculty of the Department of Computer science
California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Software Engineering

by

Eden Neba
FALL
2012
© 2012

Eden Neba

ALL RIGHTS RESERVED
RESOLVING CONFLICTING STAKEHOLDER VALUE PROPOSITIONS

IN VALUE-BASED REQUIREMENT ENGINEERING

A Project

By

Eden Neba

Approved by:

____________________________, Committee Chair
Dr. Du Zhang

____________________________, Second Reader
Bob Buckley

____________________________
Date
Student:  Eden Neba

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

__________________________, Graduate Coordinator

Dr. Nikrouz Faroughi  

Date

Department of Computer Science
Abstract

of

RESOLVING CONFLICTING STAKEHOLDER VALUE PROPOSITIONS
IN VALUE-BASED REQUIREMENT ENGINEERING

by

Eden Neba

Value based software engineering aims to incorporate economic aspects into the development of software. As a result, these economic aspects are very much integrated into the development life cycle of the software, as well as allowing different stakeholders (businesses, engineers, customers, business analyst, managers, and developers) to articulate their value propositions into the software development life cycle. When incorporating different stakeholder value propositions, there arises the issue where these different value propositions may conflict with each other because of different models that are used in coming up with these propositions. This project focuses on the stakeholder value propositions at the requirements level of the software development life cycle. The requirements level of the Software Development life cycle (SDLC) is very important because everything else builds from it. Some of the reasons for project failures are attributed to poor requirements, so to ensure a successful software development, it is important to find conflicting value propositions from stakeholders and resolve them. In this project, we first identify different conflicting requirements, and then detect various
types of inconsistent requirements based on a set of algorithms. A tool is developed that works with requirement specifications and reports the conflicting stakeholder value propositions in the value-based requirement engineering process.

_______________________, Committee Chair
Dr. Du Zhang

_______________________
Date
DEDICATION

This project is dedicated to almighty God first of all for giving me the strength and knowledge to go through this program.
ACKNOWLEDGEMENTS

Foremost, thank my God for his continue support. Many thanks go to my Dad Langsi Cletus Neba and my Mum Beltha Manka’a Neba for bringing me into this world. I also like to thank my advisors Dr. Du Zhang and Mr. Bob Buckley for their meticulous feedback, guidance and time. Their relentless support is evident by this work. I thank my family, friends and colleagues for their relentless support during my studies.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>vii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## Chapter

1. INTRODUCTION .................................................................................. 1
   1.1 Motivation ................................................................................. 1
   1.2 Problem Statement .................................................................. 2
   1.3 Document Outline .................................................................... 2
   1.4 Definition of Terms ................................................................ 3
      1.4.1 Stakeholder ..................................................................... 3
      1.4.2 Value Proposition ......................................................... 3
      1.4.3 Value-Based Requirement Engineering .............................. 3
      1.4.4 Value-Based Engineering ............................................... 3
      1.4.5 SVPs ............................................................................... 4
      1.4.6 SDLC ............................................................................... 4
      1.4.7 Arity ............................................................................... 4
2. RELATED WORK AND TECHNOLOGY ...................................................... 5
   2.1 Related Work .......................................................................... 5
2.2  Technology............................................................................................................. 8

3.  REQUIREMENT ANALYSIS ..................................................................................... 9

3.1  Conflict Types ......................................................................................................... 9

3.1.1  Complementary Conflict ..................................................................................... 10

3.1.1.1  Complementary Algorithm .............................................................................. 10

3.1.2  Mutually Exclusive Conflict ............................................................................... 11

3.1.2.1  Mutually Exclusive Algorithm ........................................................................ 12

3.1.3  Incompatible Conflict ......................................................................................... 13

3.1.3.1  Incompatible Algorithm ................................................................................. 13

3.1.4  Precedence Conflict ......................................................................................... 15

3.1.4.1  Precedence Algorithm .................................................................................... 15

3.1.5  Disagreeing Conflict ......................................................................................... 16

3.1.5.1  Disagreeing Algorithm ................................................................................. 16

3.2  General Requirement Format ............................................................................... 18

4.  DESIGN .................................................................................................................... 24

4.1  High Level Design ................................................................................................. 24

4.2  Detail Design .......................................................................................................... 27

4.2.1  File Validator ..................................................................................................... 27

4.2.2  File Parser .......................................................................................................... 27

4.2.3  Requirement Analyzer ....................................................................................... 28

4.2.4  Error Reporting ................................................................................................. 28
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1 List of Complementary Requirements</td>
<td>19</td>
</tr>
<tr>
<td>Table 2 List of Mutually Exclusive Requirements</td>
<td>20</td>
</tr>
<tr>
<td>Table 3 List of Precedence Requirements</td>
<td>21</td>
</tr>
<tr>
<td>Table 4 List of Incompatible Requirements</td>
<td>22</td>
</tr>
<tr>
<td>Table 5 List of Disagreeing Requirements</td>
<td>23</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1 High Level Design</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2 Tool Layout</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3 Processing Complementary Conflict</td>
<td>38</td>
</tr>
<tr>
<td>Figure 4 Complementary Results</td>
<td>38</td>
</tr>
<tr>
<td>Figure 5 Mutually Exclusive Results</td>
<td>39</td>
</tr>
<tr>
<td>Figure 6 Incompatible Results</td>
<td>40</td>
</tr>
<tr>
<td>Figure 7 Precedence Results</td>
<td>41</td>
</tr>
</tbody>
</table>
1.1 Motivation

In today’s uncertain and increasingly globalized business world, software companies face some challenges such as evolving new technologies, increasing global competition and dynamic market needs and customer satisfaction. To overcome some of these challenges, these companies need to achieve and maintain a competitive advantage. Customer satisfaction is extremely important to these companies, thus as a result, when these software companies are producing products, they need to ensure that the different clients (stakeholders) values are taken into consideration. Also, these companies also need to maximize their profits for a given investment during value creation process. Thus it is important for these companies to align their business strategy with technical decisions in order to drive this value. In [1] Barry Boehm states that much of software engineering today is practiced in a value neutral setting in which requirement, use case and defect are treated equally important. The value based approach in requirements engineering promotes the alignment of product, project, and business decisions [2] as well as taking into consideration the different stakeholder value propositions in the creation of the product.

In [4], Du Zhang states that most often these stakeholder value propositions are based on different models and these models conflict with each other because their underlining
assumptions are incompatible, conflicting or inconsistent with each other. Software development has many issues and many projects fail because of a variety of reasons. The reasons for the failure are many. One of the reasons for project failure is the fact that there are multiple stakeholders with conflicting value propositions. Identifying and resolving these conflicts will help move the project closer to success, thus enabling the company to achieve their goal of creating value. This project aims to develop a tool which will be used to detect conflicts between different set of requirements.

1.2 Problem Statement

Any software engineering project goes through the software development life cycle (SDLC). One of the cycles is the requirement phase. Requirements usually come from multiple stakeholders where most often than not these requirements conflict with each other. The problem being addresses in this project is to develop a tool which will automate the process of identifying the different conflicts in the requirements.

1.3 Document Outline

This section provides a brief guideline and description of topics in each chapter of the project report. In Chapter 2, we outline literature review of papers which are related to conflict identification in requirements engineering or value based engineering. Section 2.1 outlines the technology which is used in this project, namely visual studio 2008 and the programming language is C#.

Chapter 3 provides an outline analysis of the different types of conflicts which are implemented in this project. Chapter four describes the design; and Chapter five
describes implementation of the different conflicting types. Analysis of findings is covered in Chapter six. Chapter seven is the conclusion of the project. Appendix A and Appendix B contain project source code. Bibliography of references is at the end.

1.4 Definition of Terms

1.4.1 Stakeholder

A stakeholder is a person, group, organization, or system who affects or can be affected by an organization's actions

1.4.2 Value Proposition

According to [9], Value proposition is a promise of value to be delivered and a belief from the customer of value that will be experienced

1.4.3 Value-Based Requirement Engineering

According to [1], value-based requirement engineering (VBRE) aims to maximize the value of a release of software through the selection of requirements.

1.4.4 Value-Based Engineering

According to [9], value-based engineering can be defined as a software development paradigm in which required business value considerations are importantly and equally engineered into software processes, best practices, activities and tasks, management and technology decisions, tools and techniques used through the software life cycle. Value is created when a company makes profit and adding value is an economic activity that has
to be taken into account from a software perspective, since the main aim of a company is to maximize value creation for a given investment [8].

1.4.5 SVPs

Stakeholder Value Propositions.

1.4.6 SDLC

Software Development Life Cycle

1.4.7 Arity

[9] Defines the arity of a function or operation is the number of arguments or operands that the function takes. The arity of a relation is the dimension of the domain in the corresponding Cartesian product.
Chapter 2

RELATED WORK AND TECHNOLOGY

2.1 Related Work

This section is related work done by several other researchers in identifying conflicts in requirements using tools to detect these conflicts. The work that is described in this section focuses on papers written which addresses the problems of detecting and resolving conflicts in requirements.

There are many ways of identifying requirements which are in conflict with one another during the requirement phase of the life cycle of a project. This area of the software development process has drawn a lot of attention over the years and there has been a lot of studies conducted and papers written, as well as attempts to develop tools which can be used to detect conflicts in requirements. Many different methods have been used in identifying conflicts in requirements, and there is a direct correlation between system development and conflicts in requirements in that many conflicts in requirements have a negative impact on a product’s development.

In [3], Alexander Egyed and Paul Grunbacher seek to identify requirements conflicts by using the requirement attribute and requirement traceability. The requirement traceability describes and follows the life of a requirement, both backwards and forwards. The authors insist that their method can be used to find conflicting requirements at any stage of the software development life cycle as long as there are input requirements, their attributes and traces. The underlying assumption is that any two requirements conflict or
cooperate if their attributes do the same thing and trace dependency exist between them.

Their approach consist of

i) Manually categorizing the requirements into attributes

ii) Identifying cooperating and conflicting requirements

iii) Identifying dependencies among requirements

iv) Filtering out cooperation and conflict among requirements.

The only issue here is that their method is not automated.

In [4, 6] different types of inconsistencies are defined and the definition of these inconsistencies are converted into algorithms which can be used to automatically detect the conflict between the different requirements. These algorithms are easily converted into a tool which is used to automatically detect conflicts between requirements. This certainly is not the only way in which conflicts are detected between requirements.

In [7], Xuefeng Zhu and Zhu Jim state two ways of measuring inconsistencies in software requirements: qualitatively, in which case inconsistencies are analyzed based on the conditions under which they occur and quantitatively where analysis is based on calculating the degree severity of the consistency.

In [5], Alexander Egyed describes ways of automatically checking for inconsistencies during design. The method described is unique in that it does not rely on any special language nor have the rules/requirements written in a special way as required by some other tools. It allows the engineer to instantly check for inconsistencies when there is a change in design rules. This method relies heavily on detecting changes in any element in a UML diagram. If there is any change in any consistency rule in the UML diagram, the
tool developed can instantly catch this. The advantage is this method is the engineers can
detect rule changes instantly without having to wait. The author does however
acknowledge that not all consistency rules can be detected instantly.
The above mention references are just some of the methods for capturing inconsistencies
between requirements. Other school of thought considers detecting conflicts between
requirements by characterizing them at the goal level [10]. The different goals are
resolved by negotiation. [10] Uses KAOS methodology in detecting and resolving
conflicts which roughly consist of:

1) Eliciting and refining goals progressively until goals assignable to individual agents
are obtained,

2) Identifying objects and operations progressively from goals,

3) Deriving requirements on the objects/operations to meet the goals, and

4) Assigning the requirements and operations to the agents.
2.2 Technology

The technology used in this project is Microsoft Visual Studio .net. Visual Studio is an integrated IDE from Microsoft which is used to develop a variety of applications ranging from console applications to web applications and web services. There are a variety of programming languages which could be used with this IDE such as VB.NET, C#, J# just to name a few. The programming language used in this project is C# (C Sharp), which has syntax similar to Java programming language.
Chapter 3

REQUIREMENT ANALYSIS

Given a set of requirements, there could be potentially many conflicts that can exist between the different requirements. Alexander Egyed and Paul Grumbacher in [3] state that if the total number of requirements is say n, there could be up to \( n^2 \) conflicts among requirements when considering conflict among requirement attributes. As a result of the fact that there could be potentially many conflicts between requirements during the requirements phase of a software engineering process, identifying these conflicts can be a very difficult task for the engineers, especially when this has to be done manually. Also because of the numerous conflicts that can arise from requirements, there is also the possibility of false conflicts being reported. A tool to automate this process would definitely help ease the pressure on the engineers and let them focus on producing better software.

3.1 Conflict Types

This section describes and analyzes five types of inconsistencies that can exist between requirements. We can have the following types of conflicts between requirements:

i) Complementary

ii) Mutually exclusive

iii) Incompatible

iv) Precedence

v) Disagreeing
3.1.1 Complementary Conflict

A complementary inconsistency can be defined as follows:

According to Du Zhang in [4], given two stakeholder value propositions (SVPs) $V_i$ and $V_k$ and two assumptions $\alpha_i \in V_i$ and $\alpha_k \in V_k$, Let $\alpha_i$ and $\alpha_k$ be denoted by their first order formula $L_i$ and $L_k$ respectively. If $L_i$ and $L_k$ satisfy the following:

i) $L_i$ and $L_k$ contain the same predicate symbol $p$.

ii) The predicate symbol $p$ has the same arity (dimension of resulting Cartesian product) and the same terms at corresponding positions in $L_i$ and $L_k$.

iii) $L_i$ and $\neg L_k$ meaning that the truth values of the two are opposite.

When the above three conditions are satisfied, then we say that there is a complementary inconsistency or conflict between $\alpha_i$ and $\alpha_k$, denoted as $\alpha_i \neq \alpha_k$.

3.1.1.1 Complementary Algorithm

[4] defines an algorithm which can be used to determine complementary inconsistency as follows:

Let the input be denoted by $\beta$; // input requirements
Let output be denoted by $\mu \neq$; // total number of complementary conflicting cases.
Let $\mathcal{L} \neq$; denote a set of predicates for which there exit complementary literals in $\beta$. $\mathcal{L} \neq \emptyset$;
for $p \in \beta$ do
{
    If $(p(t_1, \ldots, t_n) \in \beta \land \neg p(t_1, \ldots, t_n) \in \beta)$ then
    {
        $\mathcal{E} = \mathcal{E} \cup \{p\}$;
        conflict$\neq (p) = \emptyset$;
    }
}
for $p_i \in \mathcal{E}$ do
{
    while $(p_i(t_1, \ldots, t_n) \in \beta \land \neg p_i(t_1, \ldots, t_n) \in \beta)$ do
    {
        Conflict$\neq (p_i) = $ Conflict$\neq (p_i) \cup \{p_i(t_1, \ldots, t_n), \neg p_i(t_1, \ldots, t_n)\}$;
    }
}
$\mu = \bigcup_{\rho_i \in \beta} \text{Conflict } \neq (\rho_i)$
return ($\mathcal{E}$);
return ($\mu$);

3.1.2 Mutually Exclusive Conflict

A mutually exclusive conflict defined by Du Zhang in [4] is as follows.

Given two SVPs $V_i$ and $V_k$, and two assumptions $\alpha_i \in V_i$ and $\alpha_k \in V_k$, if $\alpha_i$ and $\alpha_k$ contain literals $L_i$ and $L_k$, respectively, such that:

i. $L_i$ and $L_k$ contain different predicate symbol $p$ and $q$

ii. The predicate symbols $p$ and $q$ belong to a concept group in which predicates represent relationship that are mutually exclusive and jointly exhaustive, and

iii. $L_i = \neg L_k$ (truth values of the two are opposite);

then we say that there is an mutually exclusive inconsistency between $\alpha_i$ and $\alpha_k$, which is denoted as $\alpha_i \neq \alpha_k$. 
3.1.2.1 Mutually Exclusive Algorithm

Let Input be denoted by: \( \Omega \);

\[
\begin{align*}
\text{Pred}_{\text{mutex}} &= \{ \ldots, p, q, \ldots \} \kappa \mid (\kappa \text{ is a group of concepts}) \land (p \in \kappa) \land (q \in \kappa) \land (p \not= q) \\
\end{align*}
\]

Output: \( \exists \not= ; \) //total number of mutually exclusive conflicting cases

\( \varnothing \not= ; \) //mutually exclusive predicate pairs

\( \varnothing \not= = \emptyset ; \)

for \( \{p, q\} \subseteq \Omega \) do

\( \{ \)

if \( [(p(t_1,\ldots,t_n) \in \Omega) \land (q(t_1,\ldots,t_n) \in \Omega) \land (\delta \in \text{Pred}_{\text{mutex}}) \land (\{p, q\} \subseteq \delta)] \) then

\( \{ \)

\( \varnothing \not= = \varnothing \not= \cup \{p, q\} ; \)

\( \text{Conflict} \not= (\{p, q\}) = \emptyset ; \)

\( \} \)

for \( \{p, q\} \in \varnothing \not= \) do

\( \{ \)

while \( [(p(t_1,\ldots,t_n) \in \Omega) \land (q(t_1,\ldots,t_n) \in \Omega) \land (\{p(t_1,\ldots,t_n), q(t_1,\ldots,t_n)\} \not\subseteq \text{Conflict} \not= (\{p, q\})]] \) do

\( \{ \)

\( \text{Conflict} \not= (\{p, q\}) = \text{Conflict} \not= (\{p, q\}) \cup \{ \{p(t_1,\ldots,t_n), q(t_1,\ldots,t_n)\} \} ; \)

\( \} \)

\( \exists \not= = \bigcup_{\{p,q\} \in \rho \not=} \text{Conflict} \not= (\{p,q\}) \)

return(\( \varnothing \not= ; \));

return(\( \exists \not= ; \));
3.1.3 Incompatible Conflict

An Incompatible conflict can be defined as follows:

According to Du Zhang in [4], given a literal $L_i$ its synonymous literal $L_k$ can be defined as one that is syntactically different (having a different predicate symbol), but logically equivalent to $L_i$,

denoted by $L_i \approx L_k$. $L_i$ and its negation synonym $\neg L_k$, we use $L_i \neq \neg L_k$ to describe their incompatibility. Alternatively we can define a literal $L_k$ that is antonymous to $L_i$ as one that is syntactically different (having a different predicate symbol) and logically opposite to $L_i$ and use $L_i \neq L_k$. Given two stakeholder value propositions (SVPs) $V_i$ and $V_k$ and two assumptions $\alpha_i \in V_i$ and $\alpha_k \in V_k$, let $\alpha_i$ and $\alpha_k$ contain literals $L_i$ and $L_k$ respectively, such that

1) $L_i \approx L_k$

2) $L_i \neq \neg L_k$ where $L_i$ and $L_k$ are synonymous.

Then we say that there is incompatible conflict between $\alpha_i$ and $\alpha_k$.

3.1.3.1 Incompatible Algorithm

In [4], Du Zhang defines an incompatible algorithm as follows

Let the input literals be denoted by $\beta$:

$\text{Pred}_{\text{syno}} = \{\ldots p,q,\ldots\} | p \approx q$  \hspace{1cm} // set of synonymous predicates

$\text{Pred}_{\text{anto}} = \{\ldots r,s,\ldots\} | r \neq s$  \hspace{1cm} // a set of antonymous predicates

Let output be denoted by:

$\mu \neq$  \hspace{1cm} // a set of incompatible conflicting cases

$\xi \neq$;  \hspace{1cm} // set of incompatible predicate pairs.

$\xi \neq = \emptyset$;
for \( p \in \beta \) do

{ 
    If \([[(p(t_1, \ldots, t_n) \in \beta) \land (\neg p(t_1, \ldots, t_n) \in \beta)] \lor [\neg p(t_1, \ldots, t_n) \in \beta) \land (p(t_1, \ldots, t_n) \in \beta)] \land (\delta \in \text{Pred}_{\text{syno}}) \land ([p, q] \subseteq \delta)\] then

    { 
        \( \neg \neq = \neg \neq U \{p, q\}; \)
        Conflict\# (\{p, q\}) = \emptyset;
    }

    If \([p(t_1, \ldots, t_n) \in \beta) \land (r(t_1, \ldots, t_n) \in \beta) \land (\delta \in \text{Pred}_{\text{anto}}) \land ([p, r] \subseteq \delta)\] then

    { 
        \( \neg \neq = \neg \neq U \{p, r\}; \)
        Conflict\# (\{p, q\}) = \emptyset;
    }
}

for each \( \{p, q\} \in \neg \neq \) do 

{ 
    while \([[(p(t_1, \ldots, t_n) \in \beta) \land (\neg p(t_1, \ldots, t_n) \in \beta)] \land \{p(t_1, \ldots, t_n) \land \neg p(t_1, \ldots, t_n)\}\})

    While \([[(\neg p(t_1, \ldots, t_n) \in \beta) \land (p(t_1, \ldots, t_n) \in \beta)] \land (\neg \neq)\] do

    { 
        Conflict\# (\{p, q\}) = Conflict\# (\{p, q\}) U \{\{p(t_1, \ldots, t_n), \neg p(t_1, \ldots, t_n)\}\}
    }

    While \([[(\neg p(t_1, \ldots, t_n) \in \beta) \land (p(t_1, \ldots, t_n) \in \beta)] \land (\neg \neq)\] do

    { 
        Conflict\# (\{p, q\}) = Conflict\# (\{p, q\}) U \{\{p(t_1, \ldots, t_n), \neg p(t_1, \ldots, t_n)\}\}
    }

\[ \mu \neq = \bigcup_{pi \in \beta} Conflict \neq \{p, q\} \]

return (\( \neg \neq \));
return (\( \mu \neq \));
3.1.4 Precedence Conflict

Du Zhang in [4] defines a Precedence conflict as; given two SVPs \( V_i \) and \( V_k \), and two assumptions \( \alpha_i \in V_i \) and \( \alpha_k \in V_k \), if \( \alpha_i \) and \( \alpha_k \) assert two opposite precedence relationships for artifacts \( x \) and \( y \):

\( \alpha_i \) asserts \( \text{Precedence}(x, y) \); and \( \alpha_k \) asserts \( \text{Precedence}(y, x) \);

then we say that there is a precedence inconsistency between \( \alpha_i \) and \( \alpha_k \), which is denoted as \( \alpha_i \npreceq \alpha_k \).

3.1.4.1 Precedence Algorithm

A Precedence algorithm outlined by Du Zhang in [4] as:

Let the Input be defined by: \( \Omega \);

Let the output be defined by Output: \( \exists \npreceq \); //set of precedence conflicting cases

\[
\text{while } \left[ \left( \text{Precedence}(t_i, t_k) \in \Omega \right) \land \left( \text{Precedence}(t_k, t_i) \in \Omega \right) \land \left( \{ \text{Precedence}(t_i, t_k), \text{Precedence}(t_k, t_i) \} \not\in \text{Conflict} \right) \right] \text{ do }
\]

\[
\{ \text{Conflict} \npreceq \{ t_i, t_k \} = \{ \text{Precedence}(t_i, t_k), \text{Precedence}(t_k, t_i) \} ;
\}
\]

\( \exists \npreceq = \bigcup \text{Conflict} \npreceq \{ t, t' \} \)

return \( \exists \npreceq \);
3.1.5 Disagreeing Conflict

According to Du Zhang in [4], a disagreeing conflict is defined as follows: Given two SVPs $V_i$ and $V_k$, and two assumptions $\alpha_i \in V_i$ and $\alpha_k \in V_k$, if $\alpha_i$ and $\alpha_k$ contain literals $L_i$ and $L_k$, respectively, such that:

i. $L_i$ and $L_k$ contain the same predicate symbol;

ii. $L_i$ and $L_k$ refer to the same entity or concept; and

iii. $L_i$ and $L_k$ contain disagreeing terms: $(t_i \in L_i) \land (t_k \in L_k) \land (t_i \not\equiv t_k)$, then we say that there is a disagreeing inconsistency between $\alpha_i$ and $\alpha_k$, which is denoted as $\alpha_i \not\equiv \alpha_k$.

3.1.5.1 Disagreeing Algorithm

Du Zhang in [4] defines the algorithm for uncovering disagreeing inconsistency as follows:

Let the input be defined by: $\Omega$;

$\text{Pred}_{rei} = \{ p | p(t_i, t_j) \land p(t_i, t_k) \land (t_i \not\equiv t_k) \}$

Let the output be defined by:

$\mathcal{F} \not\equiv$    //set of disagreeing conflicting cases

$\emptyset \not\equiv = \emptyset$;

for each $p \in \text{Pred}_{rei}$ do

{  
  if $(p(t_i, t_j) \in \Omega \land p(t_i, t_k) \in \Omega \land (t_i \not\equiv t_k))$ then
  {
    $\emptyset \not\equiv = \emptyset \not\equiv \cup \{ p \}$;
    $\text{Conflict} \not\equiv (p) = \emptyset$;
    while $[(p(t_i, t_j) \in \Omega \land p(t_i, t_k) \in \Omega) \land (t_i \not\equiv t_k) \land (\{ p(t_i, t_j), p(t_i, t_k) \} \notin \text{Conflict} \not\equiv (p))]$ do
    {
      $\text{Conflict} \not\equiv (p) = \text{Conflict} \not\equiv (p) \cup \{ p(t_i, t_j), p(t_i, t_k) \}$;
    }
  }
}
\( \exists \exists = \bigcup_{p \in \wp(\exists)} \text{Conflict} \exists (\{p\}) \)

return(\(\wp(\exists)\));
return(\(\exists(\exists)\));
3.2 General Requirement Format

Every requirement can be represented in a general format, with the following components:

i. Predicate symbol: symbol which characterizes the requirement

ii. Keyword: a keyword for the requirement

iii. Requirement ID: a unique identifier for the requirement

iv. Set of attribute(s): attributes characterize the requirement e.g. response time

v. Attribute value(s): values for the attribute(s), e.g. time t=0 or t > 1

vi. Truth value: a Boolean value by which the requirement is evaluated, can evaluate to either true or false

Requirement format = \textbf{Predicate [keyword, Req id, Attribute, Attribute value(s); truth value]}

For example, consider a video on demand system with the following requirements

1) R1: Play movie automatically after selecting from list, meaning no time should elapse from choosing and playing the movie.

2) R2: One second max to start playing movie, meaning the least time that should elapse before the movie is played is one second.

The above two requirements can be represented as follows

R1 => Play [movie, R1, responsetime, t = 0; false]

R2 => Play [movie, R2, responsetime, 0 < t < 1; true]
### Table 1 List of Complementary Requirements

<table>
<thead>
<tr>
<th>Conflict type</th>
<th>Software system</th>
<th>Req ID</th>
<th>Requirement</th>
<th>Each requirement can be represented as follows: predicate symbol [keyword,req id,attribute(s),attribute value(s);truth value]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complementary</td>
<td>Software process models: Waterfall and IKIWISI</td>
<td>R1</td>
<td>Waterfall: Requirements are completely specified prior to design and implementation</td>
<td>SW[model,R1,CompReqSpec,design,impl;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2</td>
<td>IKIWISI: Complete requirements cannot be specified before the developer does some development such as prototyping</td>
<td>SW[model,R2,NotCompReqSpec,design,impl;false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3</td>
<td>Waterfall: requirements are stable once system implementation is started</td>
<td>System[req,R3,Stable,impl;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R4</td>
<td>IKIWISI: requirements are changeable during system development and evolution</td>
<td>System[req,R3,NotStable,impl;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R5</td>
<td>Waterfall: Requirements are completely documented</td>
<td>sysdoc[req,R5,compsysdoc,complete;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R6</td>
<td>Agile: requirements may not be completely documented before development</td>
<td>sysdoc[req,R6,Notcompsysdoc,complete;false]</td>
</tr>
<tr>
<td></td>
<td>Video on demand system</td>
<td>R7</td>
<td>Play movie automatically after selecting from list, i.e. at t=0</td>
<td>Play[movie,R7,resptime,t=0;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8</td>
<td>One second max to start playing a movie, i.e. at t &gt; 0</td>
<td>Play[movie,R8,resptime,t &gt; 0;false]</td>
</tr>
</tbody>
</table>
Table 2 List of Mutually Exclusive Requirements

<table>
<thead>
<tr>
<th>Conflict type</th>
<th>Software system</th>
<th>Req ID</th>
<th>Requirement</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutually exclusive</td>
<td>Video on demand system</td>
<td>R9</td>
<td>System should be portable on windows OS</td>
<td>SysWin[OS,R9, portable, windows; true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R10</td>
<td>System should be portable on mac OS</td>
<td>SysMac[OS,R10, portable, Mac; false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R11</td>
<td>System should be portable on unix OS</td>
<td>SysUnix[OS,R11, portable, Unix; true]</td>
</tr>
<tr>
<td></td>
<td>Software development model</td>
<td>R12</td>
<td>Business case: developer should implement the product in 12 months</td>
<td>Prdtime[developer,R12, developmentTime, t=12 ; true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R13</td>
<td>COCOMO model: 20 months are needed to implement the product</td>
<td>Devtime[developer,R13, developmentTime,t=20 ; false]</td>
</tr>
</tbody>
</table>
## Table 3 List of Precedence Requirements

<table>
<thead>
<tr>
<th>Conflict type</th>
<th>Software system</th>
<th>Req ID</th>
<th>Requirement</th>
<th>SWmodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precedence</td>
<td>Software implementation models</td>
<td>R14</td>
<td>Waterfall: requirements precedes implementation \Y = requirement \X = implementation</td>
<td>SWmodel[waterfall,R14,precedence,x,y;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R15</td>
<td>P=development \Y = cost \High assurance: cost precedes implementation</td>
<td>SWDev[highAss,R15,precedence,y,p;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R16</td>
<td>x=development \Y = cost \Agile: Implementation precedes cost</td>
<td>SWDev[agile,R16,precedence,p,y;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R17</td>
<td>IKIWISI: implementation precedes requirements \Y = requirement \X = implementation</td>
<td>SWmodel[ikiwisi,R17,precedence,y,x;true]</td>
</tr>
</tbody>
</table>
Table 4 List of Incompatible Requirements

<table>
<thead>
<tr>
<th>Conflict type</th>
<th>Software system</th>
<th>Req ID</th>
<th>Requirement</th>
<th>Each requirement can be represented as follows: predicate symbol [keyword,req id,attribute(s),attribute value(s);truth value]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible</td>
<td>Video on demand system</td>
<td>R18</td>
<td>Play movie automatically after selecting from list, i.e. at t=0, fast response time</td>
<td>Play[movie,R18,fastRespTime,t&gt;0;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R19</td>
<td>Runnable on a mobile device, which means slow response time</td>
<td>Play[movie,R19,SlowRespTime,t&gt;0;true]</td>
</tr>
<tr>
<td>Software development model</td>
<td></td>
<td>R20</td>
<td>Product model: unconstrained changes to the requirements, which means high cost</td>
<td>ProductModel[req,R20,HighCost, cost;true]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R21</td>
<td>Success model: knowledgeable developer means that cost should be low</td>
<td>ProductModel[req,R21,LowCost, cost;true]</td>
</tr>
<tr>
<td>Conflict type</td>
<td>Software system</td>
<td>Req ID</td>
<td>Requirement</td>
<td>ArchModel[Agile,R22,architecture,S1,cpc;false]</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Disagreeing</td>
<td>Software development models</td>
<td>R22</td>
<td>Architecture specification using agile model: the assumption is that simple design focuses on just current increment product capabilities. Cpc-current capabilities for system S1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R23</td>
<td>Product line model assumes that architecture is required for successful product line reuse. aa-application architecture for system S1</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4

DESIGN

This Chapter describes the design of the tool “Inconsistency Detector”. The Chapter consists of two parts,

1) The high level design as a general overview of the way the tool is structured and
2) Detailed design which outlines in more detail each section of the tool.

4.1 High Level Design

The figure below shows the outline of the tool, first the requirements are written in the format defined in Chapter 3. These requirements are in a pipe delimited text file. Once all the requirements are in the required format, the first thing that the tool does is to check and see if the requirements in the pipe delimited text file are all in the correct format, assuming that there is at least one requirement in the file. The requirements are then parsed into their respective components, predicate, keyword, requirement Id, attribute(s), attribute value(s), and finally truth value. These different parts are stored in variables for later use. The tool then determines which types of conflict types are to be processed (the user selects type of conflict to be processed by checking the required check box.) And based on the definitions and algorithms in Chapter 3, the parsed requirements are analyzed and results are displayed and saved in a file. If the tool finds out that there are no requirements in the file, then it exits. In case of any errors, these will be displayed on the screen as well as write them to an error file. The error file is a text document.
Figure 1 High Level Design
The figure above shows the layout of the tool. A file is uploaded for processing, and then the different types of inconsistencies to be processed are checked. Once these two conditions are satisfied, by clicking start, the tool will start analyzing the requirements.
4.2 Detail Design

The tool consists of the following main modules

4.2.1 File Validator

This module validates the file to ensure that the file format is valid before processing. If the requirement format is not valid, an error message is returned on the screen and also written to the error file.

4.2.2 File Parser

The file parser module is responsible for parsing the pipe delimited requirements into the respective components. The components are:

1) Predicate symbol: This represents a relation between the requirement and its components.

2) Keyword: the keyword represents any identifier for the requirement.

3) Requirement ID: represents a unique identifier for the requirement. This is necessary so that there is no conflict between which requirements are in conflict with each other.

4) Attribute (s): this is a defining characteristic of the requirement or a defining property of the requirement.

5) Attribute value (s): a value representing the property of the requirement.

6) Truth value: a Boolean value for the requirement. Requirements may sometime evaluate to a Boolean value.

This module also has some helper functions which aide in parsing the requirements.
4.2.3 Requirement Analyzer

This module analyzes each requirement based on their respective definitions defined in chapter 3. From the definitions, the requirement analyzer classifies the requirement as any of the conflict types defined. The conflict types are complementary, mutually exclusive, incompatible, precedence and disagreeing. Each requirement in the file is analyzed, when each line is analyzed, depending on which conflict type is selected by the user, the module calls the appropriate algorithm to determine if such requirements are in conflict with each other or not. When conflicts are discovered between requirements, the appropriate requirements IDs are displayed along with a message outline which requirements are in conflict with each other.

4.2.4 Error Reporting

This module writes any errors to an error file for later review. If a fatal error occurs (a fatal error is an error that causes the execution of the program to stop) the program terminates. The error messages are in plain English so whoever is using the tool can easily read and know what the different errors mean.
Chapter 5

IMPLEMENTATION

This Chapter consists of converting the different types of inconsistent algorithms, definitions and design into a working system. The implementation of the design in Chapter 4 was carried out incrementally by implementing module by module and one conflict type after the other. Once one conflict type was implemented, it was tested to ensure that it was consistent with the definition outlined. The method of ensuring this was to do unit testing using a set of test cases.

5.1.1 Complementary Conflict

According to Du Zhang in [4], a complementary conflict is defined such that if two SVPs have the predicate symbol p and p has the same arity and the same terms at corresponding positions and if they two SVPs have truth values that are opposite, the such a conflict is referred to as complementary. This conflict type is implemented as a function, which takes in an array of requirements, each requirement broken down into its different components, and output is a set of requirements which are in conflict with each other. Once this function is called, the array of requirements is compared, such that each requirement in the file is compared with each other. The array entry at position [i] is compared with requirement at position [i+1]. The function keeps looping until all requirements are compared with each other. According to the definition of a complementary conflict, the first thing that this function does is check to see if the requirements have the same predicates. Next is checking if the requirements have the
same elements at the same positions. The last requirement is to check and see if the requirements have opposite truth values. If all the above requirements are satisfied, the two requirements are in conflict with each other. The result from this function is stored in an array list. The reason for using an array list is because it is much easier to manipulate, adding and removing elements from it. Below is a code snippet of the implementation of this conflict type.

```csharp
public Requirements[] ComplementaryCompare(Requirements[] array1)
{
    //complementary Inconsistency
    // L1,L2 have the predicate and same arity
    // and same elements at corresponding positions in L1,L2
    //want to store the predicate from first array in a
    //variable to compare with others
    //the idea here is to compare and find out which
    //requirements have the same predicates, then
    //they belong to the same set.
    List<Requirements> req = new List<Requirements>();
    ArrayList ComplArrayList = new ArrayList();
    var strPredicate = string.Empty;
    Requirements[] ComplementaryArray;
    ComplementaryArray = new Requirements[ComplArrayList.Count];
    var prevReqID = string.Empty;
    //comp array the i and i+1 array elements
    for (var i = 0; i < array1.Length - 1; i++)
    {
        if (array1[i].Predicate.Equals(array1[i + 1].Predicate))
        {
            if (array1[i].Keyword.Equals(array1[i + 1].Keyword)
                && array1[i].Attribute.Equals(array1[i + 1].Attribute)
                && !array1[i].ReqtruthValue.Equals(array1[i + 1].ReqtruthValue))
            {
                ComplArrayList.Add(array1[i]);
                ComplArrayList.Add(array1[i + 1]);
            }
        }
    }
    var listCount = ComplArrayList.Count;
    // Get the array, convert the ArrayList into An Array
```
ComplementaryArray = (Requirements[]) ComplArrayList.ToArray(typeof(Requirements));

return ComplementaryArray;

5.1.2 Mutually Exclusive Conflict

Two events are said to be mutually exclusive if both events cannot be simultaneously true. That is if one is true, then the other cannot be. According to Du Zhang in [4], a mutually exclusive inconsistency is defined as follows: given two SVPs $L_i$ and $L_k$. If $L_i$ and $L_k$ have different predicate symbols $p$, and $q$, and if the predicate symbol $p$ and $q$ belong to a concept group in which predicates represent relationship that are mutually exclusive and jointly exhaustive and if $L_i$ and $L_k$ have opposite truth values then such an inconsistency is mutually exclusive. First the algorithm checks to see if the predicate symbols of the requirements are different. The next check is to see if the requirements belong to the same or different concept groups. Finally, if the requirements have different truth values, then all the checks are satisfied and these requirements are in conflict.

```csharp
internal Requirements[] MutexCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList mutexArrayList = new ArrayList();
    var strPredicate = string.Empty;
    Requirements[] MutexEArrayList;
    Requirements[] mylist;
    MutexEArrayList = new Requirements[mutexArrayList.Count];
    for (var i = 0; i < reqObj.Length - 1; i++)
    {
        //compare the i and i+1 element to see if they are different
        //also check to see if the attributes are different
        //they should belong to the same concept group
        if (!reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate))
```
\&\& \\
!(reqObj[i].ReqtruthValue.Equals(reqObj[i+1].ReqtruthValue)) \\
\&\& (reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword))) \\
{ 
    mutexArrayList.Add(reqObj[i]);
    mutexArrayList.Add(reqObj[i+1]);
} \\
var listCount = mutexArrayList.Count;
// Get the array, convert the ArrayList into An Array 
MutexEArrayList = (Requirements[])mutexArrayList.ToArray(typeof(Requirements));

return MutexEArrayList;

//throw new NotImplementedException();

5.1.3 Incompatible Conflict

According to Du Zhang in [4], two SVPs L_i and L_k are said to be incompatible if L_i is synonymous to L_k (that is having a different predicate symbol but logically equal), Alternatively, L_i can be defined as being antonymous to L_k, that is having a different predicate symbol and logically opposite to L_i. The implementation of this conflict type is a little bit tricky in that the value propositions can be synonymous or antonymous to each other.

In implementing this conflict type, first check is to see if the predicate symbols are different, if this is true, then we next make sure that the attribute values are the same. Any requirement that meets this condition is of an incompatible conflict type.

public Requirements[] IncompatibleCompare(Requirements[] array1)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList IncompArrayList = new ArrayList();
    var strPredicate = string.Empty;
    //Requirements[] icompatibleArrayList = new Requirements[]
    
    Requirements[] incompatibleArrayList;
5.1.4 Precedence Conflict

According to Du Zhang in [4], a Precedence inconsistency type can be defined as follows, given two $V_i$ and $V_k$, and two assumptions $L_i$ belonging to $V_i$ and $L_k$ belonging to $V_k$, If $L_i$ and $L_k$ assert two opposite precedence relationships for artifacts $x$ and $y$, then
L_i asserts Precedence (x,y) and L_k asserts Precedence (y,x). We say that there is a precedence inconsistency, between L_i and L_k. First we check to see if the predicate symbols are the same and attributes are the same and keywords are different, if these conditions hold true, then next is to check the attribute values, so see if precedence(x,y) and precedence(y,x) exist. To do this, use the new extension methods in dot net 3.0, where If the length of the Intersection of the two arrays equals that of their Union then the arrays are equal.

```csharp
internal Requirements[] PrecedenceCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList precArrayList = new ArrayList();
    Requirements[] precedenceArrayList;
    Requirements[] mylist;
    precedenceArrayList = new Requirements[precArrayList.Count];
    if (reqObj != null)
    {
        var cnt = reqObj.Length;
    }
    for (var i = 0; i < reqObj.Length - 1; i++)
    {
        //compare the i and i+1 element to see if they are different
        //also check to see if the attributes are the same
        string[] att = reqObj[i].AttributesValues.Reverse().ToArray();
        string[] attt = reqObj[i + 1].AttributesValues.ToArray();
        //Use extension methods (which are new in 3.0).
        //If the length of the Intersection of the two arrays equals that of their Union
        //then the arrays are equal
        bool equals = att.Intersect(attt).Count() ==
                        att.Union(attt).Count();
        if (reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate)
                        && reqObj[i].Attribute.Equals(reqObj[i + 1].Attribute)
                        && (!reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword)))
        {
```
if (reqObj[i].AttributesValues.Reverse().ToArray().Intersect(reqObj[i + 1].AttributesValues).Count() ==
reqObj[i].AttributesValues.Reverse().ToArray().Union(reqObj[i + 1].AttributesValues).Count())
{
    precArrayList.Add(reqObj[i]);
    precArrayList.Add(reqObj[i + 1]);
}
}

var listCount = precArrayList.Count;
// Get the array, convert the ArrayList into An Array
precedenceArrayList = (Requirements[])precArrayList.ToArray(typeof(Requirements));

return precedenceArrayList;
//throw new NotImplementedException();
}

5.1.5 Disagreeing Conflict

Du Zhang in [4] defines a disagreeing conflict as, given two SVPs \( V_i \) and \( V_k \) and two assumptions \( a \), and \( b \) belonging to \( V_i \) and \( V_k \) respectively which contain \( L_i \) and \( L_k \) respectively such that

a) If \( L_i \) and \( L_k \) contain the same predicate symbol

b) \( L_i \) and \( L_k \) contain the same entity or the concept and

c) \( L_i \) and \( L_k \) contain disagreeing terms, then we say that the requirements are of type disagreeing.

```csharp
internal Requirements[] DisagreeingCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList diagArrayList = new ArrayList();
    Requirements[] disagreeingArrayList;
    Requirements[] mylist;
    disagreeingArrayList = new Requirements[diagArrayList.Count];
    for (var i = 0; i < reqObj.Length - 1; i++)
    {
{
    // compare the i and i+1 element to see if they are different
    // also check to see if the attributes are the same
    // string[] att = reqObj[i].AttributesValues.Reverse().ToArray();
    // string[] attt = reqObj[i + 1].AttributesValues.ToArray();
    // Use extension methods (which are new in 3.0).
    // If the length of the Intersection of the two arrays equals that of their Union
    // then the arrays are equal
    // bool equals = att.Intersect(attt).Count() ==
    // att.Union(attt).Count();
    if (reqObj[i].Predicate.Substring(0, 5).ToUpper() ==
        "disgr".ToUpper())
    {
        if (reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate) &&
            reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword))
        {
            if (reqObj[i].AttributesValues.Reverse().ToArray().Intersect(reqObj[i + 1].AttributesValues).Count() !=
                reqObj[i].AttributesValues.Reverse().ToArray().Union(reqObj[i + 1].AttributesValues).Count())
            {
                diagArrayList.Add(reqObj[i]);
                diagArrayList.Add(reqObj[i + 1]);
            }
        }
    }
    var listCount = diagArrayList.Count;
    // Get the array, convert the ArrayList into An Array
    diagagreeingArrayList = (Requirements[])(reqArrayList.ToArray(typeof(Requirements)));
    return diagagreeingArrayList;
    // throw new NotImplementedException();
}
}
Chapter 6

ANALYSIS AND COMPARISON

This Chapter discusses the ways in which the implementation discussed in Chapter 5 is tested and validated. Because of the way in which the project is designed and implemented, verification and validation is done in such a way as to ensure that the software meets its specification and intended purpose. The implemented inconsistency types are tested using test cases generated from a set of requirements. These test cases are shown in Tables 1 through 5. The results from the testing are compared to known situations so as to ensure that the algorithms implemented are valid.

6.1.1 Complementary Validation

The set of test cases used for the validation of this conflict type is shown in Table 1. There are 9 different set of requirements which are used for testing and validating this conflict type. They are: R1, R2, R3, R4, R5, R6, R7, and R8. These requirements are formatted into the specified format defined in Chapter 3 and then put in a pipe delimited text file. The tool is then executed which reads the requirements in the file and processes them, the result is displayed below.
Figure 3 Processing Complementary Conflict

Figure 4 Complementery Results
6.1.2 Mutually Exclusive Validation

The cases used for validating mutually exclusive inconsistency type are R9, R10, R11, R12, and R13 as shown in Table 2. The result from executing the tool for mutually exclusive inconsistency type is shown below.

![Figure 5 Mutually Exclusive Results](image)

6.1.3 Incompatible Validation

From Table 4, the cases used for validating incompatible inconsistency are R18, R19, R20, and R21. When the tool was executed on these requirements, the result is as shown below. The requirements were compared with each other and those that are in conflict are reported.
6.1.4 Precedence Validation

Validation of precedence inconsistency type consists of the following requirements from table1. The requirements listed in Table 3 are: R14, R15, R16, and R17. When the tool was executed, the results are displayed below. Comparing the results to what was expected, both situations were the same.
6.1.5 Disagreeing Validation

The requirements used for validating disagreeing inconsistency are R22 and R23. When the tool is executed, requirements R22 and R23 are determined to be of type disagreeing. This result is compared with that obtained by manually verifying the requirement by method of inspection.
Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Inconsistencies in requirement are a significant problem in software development. Its occurrence at the requirement phase of software development life cycle shows how complex and important it is to detect and resolve them. In this project, my knowledge in detection and resolution of requirement inconsistencies has greatly increased, in part due to a lot of literature reviews, further from design and implementation of requirement inconsistency tool used to uncover inconsistencies in requirements automatically. Though there are still some manual steps involved, I believe that this project will go a long way in helping find a permanent solution to this problem. I have also increased my skills in developing applications using Microsoft.Net framework and its C# programming language. Also of importance is the fact that in designing this tool, I was able to put into use the knowledge gained from all the classes I took, such as user interface design, verification and validation just to name a few.

The work presented in this project contrast with some other work done in the area of identifying and resolving conflicts in requirements from multiple stakeholders. Holm In in his dissertation [11] is focused on identifying and resolving conflicts in requirements using quality attributes. The author developed two tools for identifying conflicts QARCC (Quality Attribute Risk and Conflict Consultant) and S-COST (Software Cost Option
Strategy Tool). This is very different from what I have done as my tool focuses on all aspects of the requirements, notably functional requirement and not just some part of it.

7.2 Future Work

Future work includes identifying and implementing additional algorithms beyond the five implemented in this tool (Inconsistency Detector). Also, there is always room for improvement as far as this tool is concerned. Further, since the requirements need to be manually put into a specified format, another future would be the ability to fully automate the process of putting the requirements into the required format.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericRequirement
{
    public class Requirements
    {
        /// each requirement has a general structure
        /// 1) A Predicate
        /// 2) A Keyword
        /// 3) A Requirement ID
        /// 4) An Attribute
        /// 5) The Attribute values
        /// 6) Truth value
        /// Example:cpl(Play Movie,R1,ResponseTime,t=0;true)
        ///      cpl(Play Movie, R2,ResponseTime,0<t<1;false)

        //accessors
        //predicate
        private string _predicate;
        public string Predicate
        {
            get { return _predicate; }
            set { _predicate = value; }
        }

        //attribute values
        private string[] _attributesValues;
        public string[] AttributesValues
        {
            get { return _attributesValues; }
            set { _attributesValues = value; }
        }

        //requirement id
        private string _reqId;
        public string ReqId
        {
            get { return _reqId; }
            set { _reqId = value; }
        }

        //operator
        private string _truthValue;
        public string ReqtruthValue
        {
            get { return _truthValue; }
            set { _truthValue = value; }
        }
    }
}
/attributes
private string _attribute;
public string Attribute
{
    get { return _attribute; }
    set { _attribute = value; }
}

private string _keyword;
public string Keyword
{
    get { return _keyword; }
    set { _keyword = value; }
}
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using AlgorithmValidator;
using GenericRequirement;
using System.Collections;

//namespace AlgorithmValidator
namespace AlgorithmParser
{
    class Parser
    {
        public Requirements[] FileValidator(string FileName)
        {
            // Read the file lines into a string array.
            // string[] strInArray = System.IO.File.ReadAllLines(FileName);

            //read everything into one string and do one loop
            StreamReader myInputFile = new System.IO.StreamReader(FileName);
            // string myInputString = myInputFile.ReadToEnd();
            // myInputFile.Close();
            //string InputArrayTemp = myInputString.Replace("\n", "");
            //strInputAarray = myInputString.Replace("\r", "");

            string[] strInputArray;
            //parse file based on pipe |

            //read everything into one line and split
            strInputArray = myInputFile.ReadToEnd().Split(new char[] {','}, StringSplitOptions.RemoveEmptyEntries);
            //array variable counter
            var j = 0;
            //declare an array to hold requirements
            Requirements[] ReqList;
            ReqList = new Requirements[strInputArray.Length];
            //or use a list to hold requirements
            List<Requirements> confList = new List<Requirements>();
            //strInputArray = InputArrayTemp.Split('|');

            foreach (string strInputArrayTemp in strInputArray)
            {
                string[] conArray;

                //instantiate a new requirement object
                Requirements req = new Requirements();
            }
        }
    }
}
string strInputArrayTemps = strInputArrayTemp.Trim();
//find index of ( , )
var index_openparam = 0;
var index_closeparam = 0;
index_openparam = strInputArrayTemps.IndexOf('(');
index_closeparam = strInputArrayTemps.IndexOf(')');

var strTempPredicate = strInputArrayTemps.Substring(0,
index_openparam);
//string temp2 =
strInputArrayTemps.Substring(index_openparam, strInputArrayTemps.Length - index_openparam);
var strTempkeyReqID =
strInputArrayTemps.Substring(index_openparam+1, (index_closeparam - index_openparam) - 1);
var strCloseParam =
strInputArrayTemps.Substring(index_closeparam,
strInputArrayTemps.Length - index_closeparam);

//parse
if (strTempkeyReqID.Length > 0)
{
    conArray = strTempkeyReqID.Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries);
    if (conArray.Length > 0)
    {
        //get array length
        var arraylength = conArray.Length;
        // reason being that the first 3 elements are
        //keyword and requirement id, and attribute
        var Noofattributes = arraylength - 3;
        req.Keyword = conArray[0];
        req.ReqId = conArray[1];
        req.Attribute = conArray[2];
        //Attributes
        string [] AttributeArray;
        //var ss = FindOccuranceOf(strTempkeyReqID,
        //',', 2);
        var ss = FindOccuranceOf(strTempkeyReqID, ',', 3);

        var truthval = FindOccuranceOf(strTempkeyReqID,
        ';', 1);
        //int ind = NthIndexOf(strTempkeyReqID, ',',
        1);
        var tval = truthval + 1;
        req.ReqtruthValue =
strTempkeyReqID.Substring(tval, strTempkeyReqID.Length - tval);
        var id = ss + 1;
        var attx = strTempkeyReqID.Substring(id,
truthval-id);
//var attxx = strTempkeyReqID.Substring(id, strTempkeyReqID.Length - id);
AttributeArray = attx.Split(new char[] { ',', }, SplitOptions.RemoveEmptyEntries);
//req.Attribute = AttributeArray[0];
req.AttributesValues = new string[Noofattributes];
for (var i = 0; i < Noofattributes; i++)
{
    req.AttributesValues[i] = AttributeArray[i];
}
}
else
{
    throw new ArgumentOutOfRangeException("length", strTempkeyReqID, "length must be > 0");
}
//req.Attribute = strTempCondition;
req.Predicate = strTempPredicate;
//confList.Add(req);
ReqList[j] = req;
j++;
}
//CompareArrayElements(ReqList);
confList = new List<Requirements>(ReqList);
//ReqList
return ReqList;

///<summary>
/// this method will be used to validate and parse the requirements for
/// Incompatible requirement types
///</summary>
public Requirements[] IncompatibleType(string fileName)
{
    //read everything into one string and do one loop
    System.IO.StreamReader myInputFile = new
StreamReader(fileName);
    string[] strInputArray;
    //declare an array to hold requirements
    //Requirements[] ReqList = new Requirements[] {};
    //parse file based on pipe |
    //read everything into one line and split
    strInputArray = myInputFile.ReadToEnd().Split(new char[] { ' | ' }, SplitOptions.RemoveEmptyEntries);
    myInputFile.Close();
    //array variable counter
    var j = 0;
    //declare an array to hold requirements
    Requirements[] ReqList;
    ReqList = new Requirements[strInputArray.Length];
//or use a list to hold requirements
List<Requirements> confList = new List<Requirements>();

foreach (string strInputArrayTemp in strInputArray)
{
    string[] conArray;

    //instantiate a new requirement object
    Requirements req = new Requirements();

    string strInputArrayTemps = strInputArrayTemp.Trim();

    //find the occurrence of the open and close parenthesis
    //find index of ( , )
    var index_openparam = 0;
    var index_closeparam = 0;
    index_openparam = strInputArrayTemps.IndexOf('(');
    index_closeparam = strInputArrayTemps.IndexOf(')');
    // compare the predicates, for Incompatible
    requirements, the predicates
    //are different
    // var strTempPredicate =
    strInputArrayTemps.Substring(0, index_openparam);
    req.Predicate = strInputArrayTemps.Substring(0,
        index_openparam);
    confList.Add(req);
}

return ReqList;

/// <summary>
/// return the index of the second occurrence of a character in a string
/// </summary>
public int FindOccuranceOf(string str, char @char, int occurance)
{
    var result = str.Select((x, y) => new { Letter = x, Index = y }).Where(letter => letter.Letter == @char).ToList();
    if (occurance > result.Count || occurance <= 0)
    {
        throw new IndexOutOfRangeException("occurance");
    }
    return result[occurance - 1].Index;
}

/// <summary>
/// return the nth index of a character in a string
/// </summary>
public int NthIndexOf(string s, char c, int n)
{ var takeCount = s.TakeWhile(x => (n -= x == c ? 1 : 0)) > 0).Count();
    return takeCount == s.Length ? -1 : takeCount;
}

///<summary>
/// compare elements in the array for the same predicates
/// this is the case for complementary requirements
/// </summary>
public Requirements[] ComplementaryCompare(Requirements[] array1)
{
    //complementary Incosistency
    // L1,L2 have the predicate and same arity
    // and same elements at corresponding positions in L1,L2
    //want to store the predicate from first array in a
    //variable to compare with others
    //the idea here is to compare and find out which
    requirements have the same predicates, then
    //they belong to the same set.
    List<Requirements> req = new List<Requirements>();
    ArrayList ComplArrayList = new ArrayList();
    var strPredicate = string.Empty;
    Requirements[] ComplementaryArray;
    ComplementaryArray = new Requirements[ComplArrayList.Count];
    var prevReqID = string.Empty;
    int cnt = array1.Length;
    //for (var i = 0; i < array1.Length; i++)
    //{
    //    strPredicate = array1[i].Predicate;
    //    for (var j = i + 1; j < array1.Length; j++)
    //    {
    //        if (strPredicate == array1[j].Predicate)
    //        {
    //            //if ()
    //            ComplArrayList.Add(array1[j]);
    //            ComplArrayList.Add(array1[i]);
    //        }
    //    }
    //}
    //}
    //compare the i and i+1 array elements
    for (var i = 0; i < array1.Length - 1; i++)
    {
        if (array1[i].Predicate.Equals(array1[i + 1].Predicate))
        {
            if (array1[i].Keyword.Equals(array1[i + 1].Keyword)
                && array1[i].Attribute.Equals(array1[i + 1].Attribute))
            {
                ComplArrayList.Add(array1[i]);
                ComplArrayList.Add(array1[i + 1]);
            }
        }
    }
}
&

!(array1[i].ReqtruthValue.Equals(array1[i+1].ReqtruthValue)))
{
    ComplArrayList.Add(array1[i]);
    ComplArrayList.Add(array1[i + 1]);
}
}

var listCount = ComplArrayList.Count;
// Get the array, convert the ArrayList into An Array
ComplementaryArray = (Requirements[])ComplArrayList.ToArray(typeof(Requirements));

return ComplementaryArray;

/// <summary>
/// this method will be used for Incompatible requirements
/// it will compare the individual requirements elements
// according to the definition of an Incompatible requirement
/// </summary>
public Requirements[] IcompatibleCompare(Requirements[] array1) {
    //want to store the predicate from first array in a
    //variable to compare with others
    //the idea here is to compare and find out which
    //requirements have the same predicates, then
    //they belong to the same set.

    List<Requirements> Req = new List<Requirements>();
    ArrayList IncompArrayList = new ArrayList();
    var strPredicate = string.Empty;
    //Requirements[] incompatibleArrayList = new Requirements[]
    
    Requirements[] incompatibleArrayList;
    Requirements[] mylist;
    incompatibleArrayList = new Requirements[IncompArrayList.Count];
    var prevReqID = string.Empty;

    int cnt = array1.Length;
    for (var i = 0; i < array1.Length - 1; i++)
    {
        //strPredicate = array1[i].Predicate;
        //compare the i and i+1 element to see if they are
different
        //also check to see if the attributes are different
        //predicates could different the same (antonymouse)
        //predicates are the samee (synonymous)
        if (!(array1[i].Predicate.Equals(array1[i +

1].Predicate))
||array1[i].Predicate.Equals(array1[i + 1].Predicate))
{
    //attributes should be different but attribute values the same
    //e.g FastResponseTime (t)/SlowResponseTime (t)
    if ((array1[i].AttributesValues[0].Equals(array1[i + 1].AttributesValues[0]))
    && (!array1[i].Attribute.Equals(array1[i + 1].Attribute)))
    {
        IncompArrayList.Add(array1[i]);
        IncompArrayList.Add(array1[i + 1]);
    }
}

var listCount = IncompArrayList.Count;
// Get the array, convert the ArrayList into An Array
incompatibleArrayList = (Requirements[])IncompArrayList.ToArray(typeof(Requirements));
return incompatibleArrayList;

// private class sortYearAscendingHelper : IComparer
//{
//    int IComparer.Compare(object a, object b)
//    {//
//        //car c1 = (car)a; car c2 = (car)b;
//        //    if (c1.year > c2.year)
//        //        return 1;
//        //    if (c1.year < c2.year)
//        //        return -1;
//        //    else
//        //        return 0;
//    }
//}
//
//icompare
public class MyReverserClass : IComparer
{
    // Calls CaseInsensitiveComparer.Compare with the parameters reversed.
    int IComparer.Compare(Object x, Object y)
    {        // you can implement this method as you wish!
        cast your x and y objects and access to their properties.
        return (new CaseInsensitiveComparer()).Compare(y, x);
    }
}

///<summary>
/// detects mutually exclusive inconsistencies
///</summary>
internal Requirements[] MutexCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
ArrayList mutexArrayList = new ArrayList();
var strPredicate = string.Empty;
Requirements[] MutexEArrayList;
Requirements[] mylist;
MutexEArrayList = new Requirements[mutexArrayList.Count];

for (var i = 0; i < reqObj.Length - 1; i++)
{
    // compare the i and i+1 element to see if they are different
    // also check to see if the attributes are different
    // they should belong to the same concept group
    if (!reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate)
        && !(reqObj[i].ReqtruthValue.Equals(reqObj[i + 1].ReqtruthValue))
            && (reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword)))
    {
        mutexArrayList.Add(reqObj[i]);
        mutexArrayList.Add(reqObj[i + 1]);
    }
}
var listCount = mutexArrayList.Count;
// Get the array, convert the ArrayList into An Array
MutexEArrayList = (Requirements[])mutexArrayList.ToArray(typeof(Requirements));
return MutexEArrayList;

//throw new NotImplementedException();
/// <summary>
/// detects precedence inconsistencies
/// </summary>
internal Requirements[] PrecedenceCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList precArrayList = new ArrayList();
    Requirements[] precedenceArrayList;
    Requirements[] mylist;
    precedenceArrayList = new Requirements[precArrayList.Count];
    if (reqObj != null)
    {
        var cnt = reqObj.Length;
    }
    for (var i = 0; i < reqObj.Length - 1; i++)
    {
        // compare the i and i+1 element to see if they are different
//also check to see if the attributes are the same
//string[] att = reqObj[i].AttributesValues.Reverse().ToArray();
// string[] attt = reqObj[i + 1].AttributesValues.ToArray();
//Use extension methods (which are new in 3.0).
//If the length of the Intersection of the two arrays
equals that of their Union
//then the arrays are equal
//bool equals = att.Intersect(attt).Count() ==
att.Union(attt).Count();

if (reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate)
    && reqObj[i].Attribute.Equals(reqObj[i + 1].Attribute)
    && (!reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword)))
{
    if (reqObj[i].AttributesValues.Reverse().ToArray().Intersect(reqObj[i + 1].AttributesValues).Count() ==
        reqObj[i].AttributesValues.Reverse().ToArray().Union(reqObj[i + 1].AttributesValues).Count())
    {
        precArrayList.Add(reqObj[i]);
        precArrayList.Add(reqObj[i + 1]);
    }
}

var listCount = precArrayList.Count;
// Get the array, convert the ArrayList into An Array
precedenceArrayList = (Requirements[])precArrayList.ToArray(typeof(Requirements));

return precedenceArrayList;
//throw new NotImplementedException();

///<summary>
/// detects disagreeing inconsistencies
///</summary>
internal Requirements[] DisagreeingCompare(Requirements[] reqObj)
{
    List<Requirements> Req = new List<Requirements>();
    ArrayList diagArrayList = new ArrayList();
    Requirements[] disagreeingArrayList;
    Requirements[] mylist;
    disagreeingArrayList = new Requirements[diagreeingArrayList.Count];

    for (var i = 0; i < reqObj.Length - 1; i++)
    {

//compare the i and i+1 element to see if they are different
//also check to see if the attributes are the same

//string[] att =
reqObj[i].AttributesValues.Reverse().ToArray();
// string[] attt = reqObj[i + 1].AttributesValues.ToArray();
//Use extension methods (which are new in 3.0).
//If the length of the Intersection of the two arrays equals that of their Union
//then the arrays are equal
//bool equals = att.Intersect(attt).Count() ==
att.Union(attt).Count();

if (reqObj[i].Predicate.Substring(0, 5).ToUpper() == "disgr".ToUpper())
{
    if (reqObj[i].Predicate.Equals(reqObj[i + 1].Predicate) && reqObj[i].Keyword.Equals(reqObj[i + 1].Keyword))
    {
        if (reqObj[i].AttributesValues.Reverse().ToArray().Intersect(reqObj[i + 1].AttributesValues).Count() !=
reqObj[i].AttributesValues.Reverse().ToArray().Union(reqObj[i + 1].AttributesValues).Count())
        {
            diagArrayList.Add(reqObj[i]);
            diagArrayList.Add(reqObj[i + 1]);
        }
    }
}

var listCount = diagArrayList.Count;
// Get the array, convert the ArrayList into An Array
diagreeingArrayList =
(Requirements[])diagArrayList.ToArray(typeof(Requirements));

return diagreeingArrayList;
//throw new NotImplementedException();
}

/// <summary>
/// class that searches for all matching objects in
/// a list
/// </summary>
/// <typeparam name="T"> public class listExt<T> : List<T>
{ //this method to retrieve all matching objects in a listEx<T>
    public T[] GetAll(T searchValue)
    {
        List<T> foundItem = new List<T>();
        for (int index = 0; index < this.Count; index++)
        {
            if (this[index].Equals(searchValue))
            {
                foundItem.Add(this[index]);
            }
        }
        return (foundItem.ToArray());
    }
}
Bibliography

The following were used to develop the methodology in this report.


11. H. In, Conflict Identification and Resolution for Software Quality Attribute Requirements, Ph.D dissertation, University of Southern California ,1998