Using Inconsistencies to Assess the Degree of Progress in Development Process

Dr. Randa Ali N. Al-Khaldi
USING INCONSISTENCIES TO ASSESS THE DEGREE OF PROGRESS IN DEVELOPMENT PROCESS

DR. RANDA ALI NUMAN KHALDI
Ajloun National Private University,
Ajloun, Jordan
randakhaldi@hotmail.com

Abstract:

In software systems, some degree of uncertainty or inconsistency is tolerated even in the final product. In such cases, there is a need to measure or evaluate even estimate the impact of these inconsistencies on software artifacts, or the frequency of failures in developing and completing the projects. However, developers often need to know the number and severity of inconsistencies in their descriptions, how various changes that they make affect these measures and if they can measure the effectiveness of development process.

In this paper, we define different algorithms for: measuring number of inconsistencies detected in each: requirements; stage of the development process and in the whole development process; degree of risk after handling; degree of progress for each cycle and each stage and in the whole development process. These algorithms help requirement engineer to revise and reanalyze his work in certain stage or decide if he had to go back and track his action, which cause such changes, consequently this will slow the development process and will increase the time remaining to accomplishing the development process, which will in its term increase the cost for fixing this problem.

Keywords: Impact on Software Artifacts, Handling Inconsistency, Degree of Risk, Managing Inconsistencies, Degree of Progress.

1. INTRODUCTION

Definitely, any changes done to a software requirements specification would create an inconsistency, which we cannot avoid during development process. If we enforce consistency this means that the change has to be delayed until the problem is sorted out, during which the desired change cannot be represented.

It is often desirable to tolerate and even encourage inconsistency [6], to maximize design freedom, to prevent premature commitment to design decisions, and to ensure all views are taken into account.

Management of inconsistency consists of a number of activities [21]:

- Detection of inconsistency which focuses on identifying specification knowledge that breaks a consistency rule.
- Classification of Inconsistency which focuses on identifying the kind of inconsistency that has been detected in a specification such as, the CONMAN system [28] which identifies six different kinds of inconsistency that may arise in programming, and uses this classification to react accordingly.
- Handling inconsistency, which focuses on acting in the presence of inconsistencies [5]. For example, when an inconsistency is detected, the appropriate action may be one of these actions: Ignore; Delay; circumvent; Ameliorate and Resolve.

So it's too important to measure the degree of impact for these inconsistencies on Software Artifacts even before deciding appropriate action for handling these inconsistencies.

2. OUR CONTRIBUTION

In this paper, we used the technique for Detecting and Locating Inconsistencies in Software Requirements through locating inconsistency and

\[ X_{ij} = \{x_{ij_1}, x_{ij_2}, \ldots, x_{ij_l}, \ldots, x_{ij_{ij_1}}, x_{ij_{ij_2}}, \ldots, x_{ij_{ij_l}}\} \]
building a matrix [14]:

\[ X_{ij} = \begin{bmatrix} R_1 & R_2 & R_j \\ r_i & 00 & 01 & ... & 10 \\ r_2 & 01 & 00 & ... & 01 \\ . & ... & ... & ... & . \\ r_i & 00 & 00 & ... & 00 \end{bmatrix} \]

Where:
\( i \) - Number of requirements
\( j \) - Number of consistency rules

This matrix consists of \((i)\) rows and \((j)\) columns for example:

\[ X_{ij} = \begin{bmatrix} R_1 & R_2 & R_j \\ r_i & 00 & 01 & ... & 10 \\ r_2 & 01 & 00 & ... & 01 \\ . & ... & ... & ... & . \\ r_i & 00 & 00 & ... & 00 \end{bmatrix} \]

Where:
\( r_i \) - Requirement with \((i)\) number
\( R_j \) – Consistency broken rule with \((j)\) number

We use this matrix to find out for each requirements \( ri = \{r_1, r_2, ..., r_i\} \) how many consistency rules each requirement broke from the sets of consistency rules
\( R_j = \{R_1, R_2, ..., R_j\} \).

2.1 MEASURING THE DEGREE OF IMPACT ON SOFTWARE ARTIFACTS

We used the algorithm for measuring degree of impact on software artifacts of inconsistencies (See Figure1) [15], to count the number of inconsistencies detected in each requirement \((NX_i)\) with the number of consistency rules broken in each requirement \((NR_j)\) and storing them in our Meta Data.

The algorithm starts with entering the number of inconsistencies in each requirement and the number of broken consistency rules in each requirement. According to these numbers we are checking the proposed impact on software artifact according to the following initial rules:

\[ di = \sum NX_i + \sum NR_j \]

Where:
\( di \) - Degree of impact on software artifacts for the \((i)\) requirement.
\( NX_i \) - Number of inconsistencies in the \((i)\) requirement.
\( NR_j \) - Number of broken rules \((R_j)\) in the \((i)\) requirement.

- If \((di)\) is less than the sum of the number of inconsistencies on the next requirement with the number of broken consistency rules in this requirement \((di \leq di+1)\) then we start to check if our \(di\) is less the \(d_{max}\) which we define it equal 100 if it is yes we keep the value of \(d_{max}\) the same and we write this \(di\) for this requirement else we change the value of \(d_{max}\) and put it equal \(di\) and we also write this \(di\) for this requirement and store it.

- If \((di)\) is greater than the sum of the number of inconsistencies on the next requirement with the number of broken consistency rules in this requirement \((di \geq di+1)\). Then we start to check if our \(di\) is less than \(d_{max}\) : if it is yes then we perform another check. If our \(di \geq d_{min}\) then we will keep the value of \(d_{min}\) the same and we write this \(di\) for this requirement, else we change the value of \(d_{min}\) and put it equal \(di\) also we write this \(di\) for this requirement. Then we are checking if the number of requirement \((i)\) less than or equal the constant \(n\) which is equal to the number of requirements stored if is yes then we enter the next requirement \( ri = ri+1\) else we stop.

2.2 ALGORITHM FOR MEASURING INCONSISTENCIES

We define a mechanism to measure the inconsistencies by counting:
1. Number of inconsistencies detected in each requirement or specification which help us further in deciding what are the actions to be taken if we had to ignore the inconsistency, or to return it to the stakeholders to rewrite it, or enforce negotiations between stakeholders to resolve, or circumvent, or ameliorate this conflict or inconsistency.

2. Number of detected times in the development process on each stage. This help us in further analyzing and deciding the degree of progress in development process accomplished which help in assessing the time for accomplishing the remaining work and its cost.

3. Number of inconsistencies detected in the whole evolving process which helps us to decide if there is missed information or a requirement is not completed or if there is no coordination between stakeholders or if there is a problem in developing process stages.

4. These results of measuring are added to the inconsistency records—Meta data for future analysis, reasoning and researches.

The proposed algorithm for measuring inconsistency shows how the process is going (see Figure 2 & 3) where:

NXi : number of inconsistencies detected in each requirement specification.
TNXc : number of inconsistencies in each cycle within a stage, TNXc = ∑ NXi
NYs : number of inconsistencies detected in each development stage and its equal the last value of TNXc for the last cycle.
NZ : number of inconsistencies detected in the whole development process, NZ=∑ NYs.
NRi : number of broken rules (Rj) in each requirement specification.
TNRc : number of broken rules (Rj) in each cycle within a stage, TNRc = ∑ NRi
SRs : number of broken rules (Rj) in each development stage and its equal the last value of TNRc for the last cycle.
TR : number of broken rules (Rj) in the whole development process TR = ∑ SRs.

Measuring started when requirements with inconsistencies entered the algorithm. For each requirement we are calculating the sum number of inconsistencies and number of consistency rules which was broken and store them for each requirement stored. Then we calculate again the sum number of inconsistencies and broken rules for each stage NYi and SRji and at the end we calculate the number of inconsistencies NZi and number of broken rules in the whole development process. All
these information are stored in Meta Rules and Data for further use, analyzing and assessing which will help also to select the action which must be taken and deciding the degree of progress in development process if it is accomplished or not and the time remaining for accomplishing the work and how it will affect the cost.

Figure 2: Algorithm of Sub-Module for Measuring Inconsistencies

2.3 ASSESSING THE DEGREE OF PROGRESS IN DEVELOPMENT PROCESS

In order to define the degree of progress we have to calculate the degree of impact on Software Artifacts \(d_i\) after handling by running the algorithm (in figure 1). Our algorithm for calculating the degree of risk after handling is shown in (figure 4 (a) & figure 4 (b)). After locating and identifying the inconsistencies after handling, we are analyzing the both values \(d_i\) before handling and after handling \(d_i'\). The algorithm starts by checking for each requirement if the both values \(d_i\) and \(d_i'\) are equal zero this means that there are no inconsistencies in this requirement before handling and after handling. But if the both values are not zeros the algorithm starts to check if

\[
\text{drisk } i' = \left[ \frac{(d_i' - d_i)}{d_i} \right] \times 100
\]

Where:
drisk _i - degree of risk on software artifact for (ri) requirement after handling.

di' - degree of impact on software artifacts for (ri) requirement with (i) number after handling.

di - degree of impact on software artifacts for (ri) requirement with (i) number before handling.

Else the degree of risk after handling will be calculated by this equation:
\[
\text{drisk } i' = ( \text{di}' - \text{di}) \times 100
\] (5)

All these values are stored in Meta Rules and Data for further use and study.

The next step after calculating the degree of risk after handling (drisk _i ') is to check:
- If the value (drisk _i ') is greater than zero then classify this requirements with its inconsistencies in group of High Risk (H).
- If the value (drisk _i ') is equal to zero then the algorithm classify this requirements with its inconsistencies in group of Moderate Risk (M).
- If the value (drisk _i ') is less than zero then the algorithm classify this requirements with its inconsistencies in group of Low Risk (L).

Also here all the values are stored in Meta Rules and Data for further use and study.

This process continue till the algorithm passes all the requirements for this cycle and then the algorithm calculate the sum of all (di) before handling and (di') after handling for all the requirements (ri):
\[
E = \sum \text{di} \quad (6) \quad \text{and} \quad F = \sum \text{di'} \quad (7)
\]

Where:
- di - degree of impact on software artifacts for (ri) requirement with (i) number before handling.
- di' - degree of impact on software artifacts for (ri) requirement with (i) number after handling.

So the degree of progress for this cycle will be calculated by this equation:
\[
\text{dprog } c = (|F - E| / E) \times 100
\] (8)

where:
- dprog _c - Degree of progress in the cycle (c).

And this value is stored in Meta Rules and Data for further use and study. After running and fishing all cycles in the stage (s) the algorithm is calculating the degree of progress for this stage by this equation:
\[
\text{dprog } s = \text{AVG} \text{dprog } c
\] (9)

Where:
- dprog _s - Degree of progress in the development stage (s).
- dprog _c - Degree of progress in the cycle (c).

After that, we run the algorithm for analyzing the degree of progress in stages (see Figure 5).

In this algorithm, we analyze if number of inconsistencies increased in one stage more than the previous stage and the degree of risk for this stage is more than the previous stage which means that we are facing a problem in this stage. This indicates requirement engineers that there is missed information in the requirements or requirement are not completed or there is no coordination between stakeholders or wrong taken actions of handling inconsistencies were done. All these reasons and others will face the requirement engineer to revise his work and reanalyze his work in certain stage or may be he had to go back and track his action which made such changes, consequently this will slow the development process and will increase the time remaining to accomplishing the development process which will in its term increase the cost for fixing this problem.

We define an Algorithm to assess the degree of progress in development process and evaluate the degree of progress between stages (see figure 5).

The Algorithm defines the following:
- If the degree of progress in stage (s) (dprog _s) is less than the degree of progress in stage (s+1) (dprog _s+1) then the degree of progress (dprog _s+1) will be classified in group of High progress which we will be denoted as (HP), and requirement engineer can continue with the next stage.
- If the degree of risk in stage (s) (dprog _s) is equal the degree of progress in stage (s+1) (dprog _s+1) then the degree of progress (dprog _s+1) will be classified in group of Moderate progress which will be denoted as (MP) and requirement engineer have to make a decision what to do next if he want to continue with this degree of progress or return and reanalyze his work again and see where he did wrong actions or decisions in stage (s+1).
- If the degree of progress in stage (s) (dprog _s) is more than degree of progress in stage (s+1) (dprog _s+1) then the degree of progress (dprog _s+1) will be classified in group of Low progress which we will be denoted as (LP), and requirements engineer definitely have to return back and reanalyze his work and see where he did wrong actions or decisions in stage (s+1).
- If the degree of progress for each stage is better than the previous stage or in some of them is equal to the next stage and upon the decision of requirement engineer the algorithm continue till it passes all the stages of development process then the algorithm calculate the degree of progress for the whole development process by this equation:

\[ \text{dprog} = \text{AVG} \text{dprog}_s \] \hspace{1cm} (10)

In our Algorithm (Figure 5) we mean by:

- **s** - Number of stage.
- **L** - Constant equal number of stage in the whole development process.
- **dprog** - Degree of progress in the development stage (s).
- **dprog**\_s+1 - Degree of progress in the development stage (s+1).

All these information are stored in Meta Rules and Data for further use and study.

Figure 4 (a): Algorithm for Calculating Degree of Risk After Handling and Calculating Degree of Progress
Yes

Calculate for all \( r_i \)

\[ \sum_{i=1}^{n} d_i = E \]

Yes

Calculate for all \( r_i \)

\[ \sum_{i=1}^{n} d_i' = F \]

Yes

Calculate degree of progress for this cycle

\[ d_{prog} = \frac{|F-E|}{E} \times 100 \]

Yes

Write the value of \( F, E \) and \( d_{prog} \)

In Meta Rules & Data

Yes

\[ C < K \]

No

calculate degree of progress for the stage

\[ d_{prog} = \text{AVG} d_{progs} \]

Yes

Write the value of \( d_{progs} \)

in Meta Rules & Data

No

\[ S \leq L \]

No

\[ d_{progs}(s) = d_{progs}(s+1) \]

Yes

Classify \( d_{progs}(s) \) in group HP

Yes

Write \( d_{progs}(s) \) for this stage in Meta Rules & Data

Yes

\[ d_{progs}(s) < d_{progs}(s+1) \]

Yes

Classify \( d_{progs}(s) \) in group MP

Yes

Write \( d_{progs}(s) \) for this stage in Meta Rules & Data

Yes

Do you want to continue with next stage

Yes

Reanalyze the stage \((s+1)\)

No

\[ s \leq L \]

Yes

Calculate the degree of progress for the development process

\[ d_{prog} = \text{AVG} d_{progs} \]

Yes

Write \( d_{progs} \) for this stage in Meta Rules & Data

No

Stop

Figure 4 (b): Algorithm for Calculating Degree of Risk After Handling and Calculating Degree of Progress

Figure 5: Algorithm to Assess and Evaluate the Degree of Progress in Development Process

3. SUMMARY AND FUTURE WORK

We defined algorithms, which calculate number of inconsistencies detected in each requirement or specification, which will help in deciding the actions that should be taken if it will be to ignore the inconsistency, or to return it to the
stakeholders to rewrite it, or enforce negotiations between stakeholders to resolve, or circumvent, or ameliorate this conflict or inconsistency. Also, we calculate the number of detected times in the development process on each stage, which will help in analyzing and deciding the degree of progress in the development process accomplished, which in turn will help in assessing the time for accomplishing the remaining work and its cost. Calculated number of inconsistencies detected in the completely evolving process will indicate software engineers about the frequency of failure in the development process in order to assess the system reliability and the degree of progress in development process and its effectiveness.

Future work is needed to evaluate how the degree of progress reflects in time and costs. Also, to build a knowledge base system to direct management of inconsistency process by building a machine learning system. The machine learning system will take use of all collected and stored information in Meta Data for automated reasoning; learning from previous experiences; statistics; pattern analysis and further data mining.

REFERENCES: