Abstract

Context: The dependencies between individual requirements have an important influence on software engineering activities e.g., project planning, architecture design, and change impact analysis. Although dozens of requirement dependency types were suggested in the literature from different points of interest, there still lacks an evaluation of the applicability of these dependency types in requirements engineering.

Objective: Understanding the effect of these requirement dependencies to software engineering activities is useful but not trivial. In this study, we aimed to first investigate whether the existing dependency types are useful in practise, in particular for change propagation analysis, and then suggest improvements for dependency classification and definition.

Method: We conducted a case study that evaluated the usefulness and applicability of two well-known generic dependency models covering 25 dependency types. The case study was conducted in a real-world industry project with three participants who offered different perspectives.

Results: Our initial evaluation found that there exist a number of overlapping and/or ambiguous dependency types among the current models; five dependency types are particularly useful in change propagation analysis; and practitioners with different backgrounds possess various viewpoints on change propagation. To improve the state-of-the-art, a new dependency model is proposed to tackle the problems identified from the case study and the related literature. The new model classifies dependencies into intrinsic and additional dependencies on the top level, and suggests nine dependency types with precise definitions as its initial set.

Conclusions: Our case study provides insights into requirement dependencies and their effects on change propagation analysis for both research and practise. The resulting new dependency model needs further evaluation and improvement.

Key words: requirement dependency, requirement traceability, requirement relationship, change propagation, impact analysis, case study

1. Introduction

In developing software intensive systems, most individual requirements relate to and affect each other in complex manners [26]. Various and complex dependencies among requirements increase the difficulties in understanding the requirements and influence many software engineering activities such as architecture design [18], product release planning [6] and change impact analysis [27]. Requirement dependencies are also important inputs for component selection [14] and web service composition [31].

In recent volatile environment, software systems must evolve to adapt themselves to the rapid changes of stakeholders’ needs, technologies and business environments [20]. To control the risks brought by software evolution, it is vital to analyse change propagation in order to determine what other parts of a software may be affected if a change is made. Evolution of software systems is mostly studied at the level of code and design with a focus on code reengineering/migration, architectural evolution and software refactoring [1, 34]. However, it is also necessary to analyse change propagation earlier at the requirements level. This kind of change propagation analysis can provide important change-related information from a business point of view. Requirements-level change propagation analysis has also been regarded as one important area in software evolution research [13].

Requirements dependency is the relationship between requirements and acts as the basis for change propagation analysis. Dozens of dependency types have been proposed to reflect the complex relationships between requirements at both structural and semantic levels. These dependency types...
types have different levels of abstraction and are used in various aspects of project management. Pohl [24], Dahlstedt and Persson [9] proposed the dependency types respectively based on literature survey in the areas of requirements engineering. Karlsson et al. introduced the dependency types to prioritise requirements [17]. Carlshamre et al. presented a study of requirements dependencies in release planning [7].

However, there still lacks an evaluation of the applicability of these dependency types in a real-world project and their effectiveness in change propagation analysis. This limits the wide use of these dependency types for both dependency identification and change propagation analysis. In practise, researchers and practitioners usually choose from these dependency types based on their own experiences and ignore other dependencies that may propagate changes. As a result, the accuracy of the change propagation result may decrease. In one of our early studies [23] we chose three dependency types arbitrarily from Dahlstedt’s dependency model [9] to estimate the change impact at the code level. This choice ignored other dependency types in Dahlstedt’s dependency model that may have propagated changes and subsequently influenced our change propagation estimation. In addition, some change impact analysis research merely focuses on change propagation in a specific requirements model, such as use case map model [15], which severely constrains the range of dependency types and causes change propagation analysis incomplete.

Given the very limited evaluation endeavour on requirements dependency, our research aims to empirically investigate the usefulness and applicability of existing dependency models (types), and further propose the improvements that are based upon the evaluation results to effectively facilitate dependency identification and change propagation analysis. To achieve the goals, we conducted a case study with three participants to evaluate the usefulness and applicability of existing dependency types in a real-world industry project. The evaluation objects are 25 dependency types defined in Dahlstedt’s dependency model (D-model) [9] and Pohl’s dependency model (P-model) [24], which are the two most adopted dependency models in software requirements. This case study is aimed to answer four research questions:

**RQ1** What dependency types are (not) used to identify relationships between requirements and why?

**RQ2** What dependency types are (not) used in change propagation analysis and why?

**RQ3** How effective/ineffective are dependency types used?

**RQ4** What are the main factors affecting the discovery of dependencies?

Corresponding with these research questions, the evaluation presented in this article has found that: 1) seven dependency types in the P-model and three dependency types in the D-model were deemed applicable by the practitioners to describe relationships between requirements; 2) five dependency types can indicate change propagation particularly well, but their definitions need to be clarified; 3) dependency types are helpful to find more dependencies, but some dependency types have ambiguous definitions or overlap with each other which increases the difficulty of use; 4) four main factors affecting the discovery of dependencies. We also find participants with various backgrounds have different viewpoints on change propagation. Change impact analysis should involve a wide range of stakeholders including project managers, requirements engineers, designers and developers. Moreover, we provide a group of specific dependency types for change propagation analysis based on our empirical findings.

Based upon our empirical evaluation of existing dependency models and the critique of current dependency types, we further propose a new dependency model by refining, integrating and extending the previous ones. The new dependency model classifies dependencies into intrinsic dependencies and additional dependencies on the top level. The former reflects essential dependencies existing among requirements, which are more likely to impact many various software engineering activities; the latter represent relationships imported by certain software engineering tasks. The initial set of the new model includes nine generic dependency types.

This article is the extended version of the conference paper published in EASE 2012 [22]. Compared to the original version, the major extensions in this article are the detailed description of the case study design, execution, and data analysis (Section 3 & 4), as well as the presentation of a new dependency model (Section 5) based on the findings from the case study. In addition, this article also include numerous minor updates, corrections, and enhancements to the conference paper.

The rest of this article is structured as follows. Section 2 introduces the background and the related work about requirements traceability, requirement dependency, and change propagation analysis, and also briefly compares the existing dependency types whose problems motivated our research. Section 3 describes the context and method of the case study for evaluating the existing dependency types. The results and analysis of the case study are reported in Section 4 for answering the research questions in terms of usefulness and applicability. Section 5 proposes a new dependency model and redefines a set of common dependency types based on our empirical findings. Section 6 discusses the dependency types for change propagation analysis in particular as well as the limitations of our study at this stage. We conclude our research with suggestions for further work in Section 7.
2. Background and Motivations

2.1. Dependency in Requirements Engineering

In order to analyse the impact of a proposed software change, it is vital to determine which parts of the software system may be affected by the change and ascertain their possible risks [5]. Change propagation means that a change can impact on not only source code, but also other software artifacts, such as requirements, design and test cases [4]. Bohner proposed an impact analysis process [5] which examines the change requests to identify the Starting Impact Set (SIS) of software artifacts that could be affected by the required change. Artifacts in the SIS are then analyzed to identify other artifacts anticipated to be affected. The newly identified artifacts are incorporated into SIS to form the Candidate Impact Set (CIS). On the other hand, an Actual Impact Set (AIS) consists of the set of artifacts actually modified after the change is implemented. The goal of the impact analysis process is to estimate a CIS that is as close as possible to the AIS. Along this topic, Fasolino and Visaggio [12] showed how CIS can be determined step by step starting from the highest-level documentation (e.g., requirements specification) affected by the changes down to the source code to be changed. This top-down analysis approach can be efficiently supported through traceability information [10].

Traceability research is gaining an increasing attention in many areas such as requirements engineering and model driven architecture [29]. A traceability link is “any relationship that exists between artefacts involved in the software engineering life cycle” [2]. It includes not only the forward and backward links between artefacts (e.g., requirements and architecture, requirements and code), but also links between items within a software development artefact (e.g., requirements dependency, code dependency). Correct traceability is the basis for change propagation analysis [27] which is important for all aspects of a software development project.

Many existing research studies explore how to discover traceability automatically. For example, information-retrieval technology is used to identify the relationships between documentation and code [3], between requirements and requirements [16], and between requirements and test cases [11]. However, solutions are far from mature and practitioners still need to manually identify or confirm dependencies especially for requirements due to the rich semantics in natural language based requirements.

To help people identify relationships between requirements, many (inter-)dependency models (e.g., [24, 17, 7, 9]) have been proposed to assist in identifying and classifying requirements relationships into dependency types based on the structural and semantic properties of requirements. Whereas, there has been no empirical evaluation of any these dependency types within real world software projects in terms of the degree of usefulness and applicability.

Requirements dependencies also play an important role in change propagation analysis. Hassine et al. applied dependency analysis at the use case map level (rather than between requirements in natural languages) to identify the potential impact of requirement changes on the overall system [15]. Khan et al. developed a concern-oriented dependency taxonomy to capture the relationships between requirements-level concerns and their manifestation at the architectural level [19]. Yan et al. discussed the ripple-effect of requirements evolution based on requirements dependency [30]. Though all these studies evaluated the generic dependency type with some definitions loosely related to the dependency types defined in the Pohl’s [24] and Dahlstedt’s [9] dependency models, there is no empirical evaluation of the effectiveness of dependency types defined in the both models in terms of change propagation analysis.

2.2. Requirements Dependency Models

Many dependency types have been proposed in various dependency models over years and most of them have different levels of abstraction and different criteria for categorisation. This has increased the difficulty for evaluation. Among these models, there are two representative generic requirement dependency models. These models suggest 25 dependency types to categorise different characteristics of relationships between requirements. The dependency model proposed by Pohl [24] in 1996 was based on a survey of over thirty publications in the area of requirements engineering (Figure 1). The other is a requirement (inter-)dependency model proposed by Dahlstedt and Persson [9] in 2005 (Figure 2). The dependency types evaluated in this study come from these two well-known dependency models.

![Figure 1: Pohl’s dependency model](image-url)

The Pohl’s dependency model (P-model) includes 18 dependency types based on a literature survey. The Dahlstedt and Persson’s interdependency model (D-model) proposes 7 dependency types which combine some existing definitions (including [24, 17, 6, 7, 25, 27]) into an integrated view. According to the definitions in the D-model, some dependency types in the P-model overlap with and encompass the dependency types in the D-model.
(as shown in Table 1). The semantics of some dependency types in the D-model is more general than the P-model. In our study, we do not differentiate these overlapping dependency types because we want to identify which dependency types are more applicable in practical use.

Table 1: Dependency overlapping between D-model and P-model

<table>
<thead>
<tr>
<th>D-model</th>
<th>P-model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explains</td>
<td>Requires</td>
</tr>
<tr>
<td>Replaces</td>
<td>Precondition</td>
</tr>
<tr>
<td>Based_on</td>
<td>Similar_to</td>
</tr>
<tr>
<td>Formalises</td>
<td>Conflict_with</td>
</tr>
<tr>
<td>Elaborates</td>
<td>Constraint</td>
</tr>
<tr>
<td>Generalise</td>
<td>Conflicts</td>
</tr>
<tr>
<td>Refines</td>
<td>Refined_to</td>
</tr>
<tr>
<td>Temporal</td>
<td>Similarity</td>
</tr>
<tr>
<td>Requires Precondition</td>
<td>Resource</td>
</tr>
<tr>
<td>Similar_to</td>
<td>Similarity</td>
</tr>
<tr>
<td>Constraint</td>
<td>Reference</td>
</tr>
<tr>
<td>Conflicts</td>
<td>Satisifies</td>
</tr>
<tr>
<td>Refined_to</td>
<td>Generalise</td>
</tr>
</tbody>
</table>

In addition, some existing literature tends to address requirement dependency (or relationship) based on specific problems, e.g., requirement prioritisation [17], software product release planning [7], requirement recycling [28]. Although the existing research proposes dependency types from different perspectives, we observe there is still a large overlapping in all these dependency (or relationship) models. Table 2 shows the dependency types we classified from the above mentioned models according to their uses for project and product related activities, in which many similar dependency types appear in several models.

Table 2: Dependency types vs. software engineering activities

<table>
<thead>
<tr>
<th>SE activity</th>
<th>Related requirement dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture design</td>
<td>Constraint, Conflicts[24]; Conflict_with[9]; verb categorisation related relationship types to identify crosscutting dependency[8]</td>
</tr>
<tr>
<td>Test</td>
<td>Precondition[24]; Requires[9, 7]; Temporal, And[25, 7]; Task, Causality[20]</td>
</tr>
<tr>
<td>Change impact analysis</td>
<td>Similar[24]; Similar_to[9]; Structure, Resource[25]; Similarity, Reference[23]; Refines[24]; Refined_to, Generalise[9]</td>
</tr>
<tr>
<td>Requirement quality checking</td>
<td>Compares, Contradicts[25]; Example_for, Test_case_for, Purpose, Background, Comments[24]</td>
</tr>
<tr>
<td>Requirement evolution</td>
<td>Replaces, Satisfies, Based_on, Formalises, Elaborates, Changes_to[24]</td>
</tr>
<tr>
<td>Product releasing</td>
<td>Increase/Decrease_cost_of, Increase/Decrease_value_of[9]; Cannot_exist, Must_exist, Positive_cost, Negative_cost, Positive_value, Negative_value[17]; Requires, And, Temporal, Or, CValue, ICost[7]</td>
</tr>
</tbody>
</table>

2.3. Motivations

The above review, comparison and analysis imply an apparent need for a basic and common set of dependency types among requirements with clear definitions, and inspire us to develop an integrated dependency model. To achieve this, we first need to obtain an overall understanding about and in-depth analysis of requirement dependencies and their use in real world software projects. Nevertheless, the current literature still lacks related empirical research on this topic. The case study reported in this article provides an initial evaluation of the usefulness and applicability of the typical requirement dependency types in a real-world industry project.

Based upon the existing dependency models in the literature, in particular the initial evaluation of the P-model [24] and the D-model [9], we next propose a new dependency model to define the links between different relationships at the concept level. This model may help to reduce the ambiguity and overlapping between the existing models by providing clear and precise definitions.

3. Case Study: Method

The goal of the case study is to empirically evaluate the usefulness and applicability of all the dependency types defined in the two well-known dependency models (P-model and D-model) in terms of both dependency identification and change propagation analysis.

3.1. Context

The Lending Industry XML Initiative (LIXI) is an independent non-profit organisation that has been established to remove data exchange barriers within the Australian lending industry. Through the work of LIXI, member organisations are able to provide services to their customers more efficiently and at lower cost. This is achieved by establishing an open XML standard for the format and exchange of lending-related data that replaces numerous incompatible and proprietary approaches. Members of LIXI come from a broad range of companies and organisations across the lending industry in Australia. They include major banks, mortgage originators and brokers, mortgage insurers, property valuers, settlement agents, trustees and information technology providers. In 2008, NICTA developed a property valuation system (PVS v2.0) for a company in the LIXI organisation using the LIXI standards [32, 33]. This system includes a ‘Pocket Valuer’ sub-system (running on PDA) for capturing property valuation data in field, a desktop sub-system for managing information and a web-based business process system for managing the workflow. Now, the company is making plans to move all the desktop sub-system to a web-based system and implement some new functions. In this paper, this new system is called PVS v3.0. This study is conducted as part of the PVS v3.0 requirements development and system design project. Our study uses requirements of PVS v2.0 for dependency identification as well.
as change propagation analysis triggered by new requirements in PVS v3.0.

3.2. Method

The case study was designed and conducted in order to answer the four research questions (cf. Section 1).

3.2.1. Participants

Three practitioners (A, B, and C) from NICTA were involved in this case study. Participant A is the project manager and the architect of the PVS systems who had eight years of experiences with project management and software architecture. Participant B is a new requirements engineer/researcher who was not familiar with the system very much but had five years of practical experience of requirements analysis and management. Participant C is a developer with two years of software development experiences who knew the system well.

Due to the nature and the size of the project, three participants came with different backgrounds. Although such difference may impact the case study results, this selection gave us the opportunity to explore whether different backgrounds affect the dependency identification and the change propagation analysis.

3.2.2. Execution

The case study selected the response variables in corresponding to the research questions: number of dependency types and dependencies identified (RQ1), number of dependency types and dependencies used in change propagation analysis (RQ2), efficiency, correctness and distribution of dependency types and dependencies (RQ3 and RQ4).

Prior to the execution of the case study, the three participants received an introduction about requirements traceability, requirements (inter-)dependencies, and change propagation analysis. When the case study started, each participant was given the following materials: 1) PVS v2.0 requirements document in natural language (144 requirements in 19 modules), 2) PVS v3.0 requirements document (33 requirements in 9 modules), 3) matrix for dependency identification, and 4) original D-model and P-model documents.

The participants first learned the requirement dependency types by reading the materials (D-model and P-model documents). With reference to the documents of the two dependency models, the participants read the requirements specification of PVS v2.0, as well as discovered dependencies in the requirements of the system and identified their corresponding dependency types individually. The identified dependencies were recorded in a dependency matrix. Next in change propagation analysis, they manually analysed the requirements (PVS v2.0) estimated to be changed because of the change requests from the new requirements of PVS v3.0, and recorded the identified change propagation paths.

The study was conducted at NICTA Sydney Lab, and the whole process was under supervision. The time spent on the case study (including self-learning, dependency identification, and change propagation analysis) was not limited. When they finished, they were asked to complete a questionnaire to report problems and their experience, as well as evaluate the two dependency models.

3.2.3. Data Collection and Analysis

The direct outputs of this case study that were collected in the end individually are: 1) all dependencies identified in PVS v2.0 requirements (in the form of dependency matrix), 2) change propagation paths and dependency types used in change impact analysis. Initial rationale and comments were collected from each of the participant as a part of their exercise.

The usefulness and applicability of the dependency types in both identification and change propagation were analysed across the three participants. The follow-up questionnaire was used to elicit in-depth reasons behind the differences and the similarity. Common consensus was reached on all issues.

The detailed data we intended to collect are shown in Table 3. These data are linked to research questions and collected through a dependency matrix, lists of dependency and propagation path identified, and a questionnaire. The collected data was in quantitative (mainly for RQ1 and RQ2) and qualitative (for RQ3 and RQ4) formats. The quantitative data from each participant was analysed individually, then compared across the three. The questionnaires completed by the participants in the end of the case study capture the data for answering RQ3 and RQ4. After the case study, the three participants were interviewed individually to collect rich qualitative information regarding RQ3 and RQ4, as well as any issues and concerns raised in the questionnaires.

![Table 3: Data collection for research questions](http://itechs.iscas.ac.cn/en/products/downloadfiles/case_study_data.rar)

4. Case Study: Results and Analysis

This section presents our initial analysis of the case study results\(^1\) for answering the four research questions.

\(^1\)The raw data of the case study is available at [http://itechs.iscas.ac.cn/en/products/downloadfiles/case_study_data.rar](http://itechs.iscas.ac.cn/en/products/downloadfiles/case_study_data.rar).
4.1. Dependency types in relationship identification (RQ1)

Dependency types used by three participants are listed in Table 4, which clearly shows the usage of all dependency types in this case study. We found the uses of some dependency types were not limited to their original definitions and participants might possess their own understandings about some dependency types, in particular on the following dependency types.

Table 4: Dependency types found (not found) in PVS v2.0

<table>
<thead>
<tr>
<th>Dependency type</th>
<th>Found by</th>
<th>Reason (if unused)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>Precondition</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>Similar_to</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>Similar</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td>A B</td>
<td></td>
</tr>
<tr>
<td>Satisfies</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Contradicts</td>
<td>- B</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>- B</td>
<td>never found</td>
</tr>
<tr>
<td>Based_on</td>
<td>- B</td>
<td>never found</td>
</tr>
<tr>
<td>Refines</td>
<td>- C</td>
<td>never found</td>
</tr>
<tr>
<td>Refined_to</td>
<td>- C</td>
<td>never found</td>
</tr>
<tr>
<td>Compares</td>
<td>- A C</td>
<td>hard to understand</td>
</tr>
<tr>
<td>Conflicts_with</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Change_to</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Example_for</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Test_case_for</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Purpose</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Background</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Increase/Decrease_cost_of</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Increase/Decrease_value_of</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Replaces</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Elaborates</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Formalises</td>
<td>- -</td>
<td>never found</td>
</tr>
<tr>
<td>Generation</td>
<td>- -</td>
<td>never found</td>
</tr>
</tbody>
</table>

- **Similar_to** is more precise than **Similar**. **Similar** means “two objects are similar”, while **Similar_to** means “one stated requirement is similar to or overlapping with one or more other requirements”. Although in the D-model **Similar_to** is regarded the same as **Similar**, in the case study, three participants preferred **Similar_to** because its definition is clearer and easier to use. **Similar_to** exists between requirements which describe different operations on the same objects, such as ‘add valuation information’, ‘delete valuation information’ and ‘modify valuation information’.

- **Precondition** is identical with **Requires**. They are often used to describe the process control relationships. In the P-model and the D-model, **Requires** and **Precondition** are used to describe the hierarchical relationship between two requirements. In this case, for example, ‘the ability to create valuation requests’ is the precondition of ‘the ability to delete valuation requests’. Here, **Precondition** is the most identified dependency in individual modules and between modules.

- **Refines** is the alternative to **Refined_to**. They are used to describe the relationships between require-

ments at the same level. **Refines** means “a requirement is defined in more detail by another requirement”, and **Refined_to** means “a high-level requirement is refined by a number of more specific requirements”. In the D-model and the P-model, these two dependency types describe the hierarchical structure of requirements. For example, in one function module in this case, ‘set user permissions’ refines ‘only permit users to view/edit/delete data that they have the correct permissions for’.

- **Constraint** can be used to describe the relationships between non-functional requirements and functional requirements. Constraint links are used to “relate a constraint to a particular object”. But what is ‘a constraint’ is not clearly explained in the P-model.

- **Satisfies** describes a specific kind of constraint relationship between requirements. **Satisfies** expresses that “if the target object is reached by the final system, the source object is also satisfied”. For example, in the PDA system, one requirement ‘sign off screen (with signature capture)’ in the desktop system satisfies another requirement ‘the valuer must submit their valuation for authorisation once they are complete’.

Table 4 also indicates some dependency types never used (found) in the case study. According to the participants’ feedback, the reasons why these dependency types were not used might be:

1. Some dependency types are hard to understand.

- **Compares** describes “the result of a comparison and the objects which have been compared”. This definition does not state the use of this dependency type and is hard to understand.

- **Conflicts** indicates that “fulfilling one requirement has negative influence on the other one”. But the definition does not state clearly what ‘negative influence’ is. Two participants found this dependency, but their understandings were different. One regarded ‘negative influence’ as the requirements could not be implemented and the other understood it as the requirement would conflict with other requirements. So its definition needs to be clarified.

2. Some dependency types could not be found in requirements.

- **Example_for**, **Test_case_for**, **Background** and **Purpose**. There are no related contents in the requirements document.

- **Increase/Decrease_value_of** and **Increase/Decrease_cost_of**. Given their original definitions, these two dependency types are hard to judge in requirements document.
• Evolutionary links in the \( P \)-model, e.g., \textbf{Replaces}, \textbf{Elaborates} and \textbf{Based_on}. These dependency types record the change history of requirements. There are no such dependencies in PVS v2.0 requirements (discussed in the next subsection).

4.2. \textit{Dependency types in change propagation analysis (RQ2)}

There are two steps in change propagation analysis process used by three participants: direct change propagation and indirect change propagation. The first step is to identify PVS v2.0 requirements directly impacted by PVS v3.0 requirements. The next step is to identify requirements impacted indirectly in PVS v2.0.

1. \textbf{Replaces} and \textbf{Elaborates} were used in the direct change propagation analysis.

Two participants used these two types to judge how PVS v2.0 functional requirements were impacted by PVS v3.0 requirements. \textbf{Replaces} and \textbf{Elaborates} belong to \textit{evolutionary} links in the \( P \)-model which describe how requirements evolve into their new versions. In this case, \textbf{Replace} results in a wider impact scope than \textbf{Elaborate} because if the new requirement replaces the old one, the old function may be discarded.

• \textbf{Replaces} are used to express a particular requirement (source object) has been replaced by the target object. In this case, some desktop functions in PVS v2.0 need to be migrated to web system. This change can be identified as new (web) requirements replace old (desktop) requirements.

• \textbf{Elaborates} typically means the new requirement improves another previous requirement. In this case study, some functions are to be upgraded between versions of the system. For example, the valuation request can only be created manually in PVS v2.0, but in PVS v3.0 the valuation request can be created automatically from PDF file. This kind of change scenario means the old functions have to be refined.

2. \textbf{Precondition}, \textbf{Similar_to}, \textbf{Constraint} were used in the indirect change propagation analysis.

Table 5 shows the dependency types used by three participants. There are 19 modules in the PVS v2.0 requirements document noted as \( M_1^{\text{-}M_{19}} \) and 9 modules in the PVS v3.0 noted as \( M_1^{\text{-}M_{9}} \). Module means a coherent set of functional or nonfunctional requirements. The analysis results of three participants are different and there exists overlapping between them. People’s viewpoints and experiences may affect the change impact analysis. Participant A found most requirements impacted because he is familiar with the business process and system implementation. When A was analysing the change propagation, he considered if the change would happen in design and code factually. B found the requirements impacted from business point of view, so she found some requirements whose implementations are not necessarily to be changed but her findings are helpful to improve the design. C analysed the change propagation from the development point of view. The requirements impacted found by C are to be changed in code.

We summarise the change propagation examples of the above three dependency types.

• \textbf{Similar_to} exists between many functional requirements dealing with the same data information. If the data information is to be changed, all the related similar functions may be changed too. For example, as shown in Figure 3, there is a new requirement that is \textit{‘the system can create a new valuation request from PDF requests from an email account or a file system’} in PVS v3.0. This requirement impacts all the functions dealing with the request information in PVS v2.0 such as \textit{‘user can view valuation requests manually’} and \textit{‘user can cancel valuation requests’} because the user interfaces of these functions may need to add the ‘PDF’ link for users to check the PDF document.

![Figure 3: ‘Similar_to’ in change propagation](image)

• \textbf{Constraint} exists between non-functional requirements and functional requirements. For example, Security module includes requirements \textit{‘system login’} and \textit{‘only permit users to view/edit/delete data that they have the correct permission for’}. Security module constrains all the functions which should be accessed by users with privileges. If some functions’ privilege constraints are changed, then the security module may need to be modified. For instance (Figure 4), in PVS v2.0, only the manager can see and authorise valuation, but in PVS v3.0 the administrator is also permitted to see and authorise valuation. So \textit{‘valuation authorisation by manager’} and the Se-
PDA submits valuation
Quality checking on web
Impact
Precondition Change propagation
quality checking on PDA

Table 5: Change propagation analysis

<table>
<thead>
<tr>
<th>Requirements in PVS v3.0</th>
<th>Participant</th>
<th>Requirements module impacted directly</th>
<th>Requirements module impacted indirectly</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₃: Auto-processing request from PDF document (new function)</td>
<td>A</td>
<td>M₇: Precondition/Require</td>
<td>M₇: Similar/Similar_to, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>M₆: Precondition/Require</td>
<td>M₆: M₆₈, Similar/Similar_to, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>M₇: Precondition/Require</td>
<td>M₇: Similar/Similar_to</td>
</tr>
<tr>
<td>M₄: The Request-related desktop features shall be migrated to the Web (upgrade)</td>
<td>A</td>
<td>M₆₈, M₆₇, M₆₈, M₆₉: Similar/Similar_to, Precondition/Require</td>
<td>M₆₈: Similar/Similar_to; M₆₇: Satisfies, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>M₆₈, M₆₇</td>
<td>M₆₈: Satisfies, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>M₇</td>
<td></td>
</tr>
<tr>
<td>M₅: Automatically allocate a valuation task to a valuer according a set of rules (new function)</td>
<td>A</td>
<td>M₇, M₇₄, M₇₅, Constraint, Precondition/Require</td>
<td>M₇₄: Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>M₇</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>M₇</td>
<td></td>
</tr>
<tr>
<td>M₆: The Booking-related desktop features shall be migrated to the Web (upgrade)</td>
<td>A</td>
<td>M₇, M₇₆, M₇₇, M₇₈: Similar/Similar_to, Precondition/Require</td>
<td>M₇₆: Similar/Similar_to, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>M₇, M₇₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>M₇</td>
<td></td>
</tr>
<tr>
<td>M₇: Valuation report quality assurance-PDA (new function)</td>
<td>A</td>
<td>M₇₆, M₇₈, M₇₉: Similar/Similar_to, Precondition/Require</td>
<td>M₇₆, M₇₈: Similar/Similar_to, Precondition/Require</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>M₇₆, M₇₈</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>M₇</td>
<td></td>
</tr>
<tr>
<td>M₈: The new features shall be integrated with V1 features through a configurable business process engine for ease of change in the future (upgrade)</td>
<td>A</td>
<td>All the requirements</td>
<td>All the requirements</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>All the requirements</td>
<td>All the requirements</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>All the requirements</td>
<td>All the requirements</td>
</tr>
<tr>
<td>M₉: Upgrade the remaining of the desktop features to web application (upgrade)</td>
<td>A</td>
<td>All the desktop system requirements</td>
<td>All the desktop system requirements</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>All the desktop system requirements</td>
<td>All the desktop system requirements</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>All the desktop system requirements</td>
<td>All the desktop system requirements</td>
</tr>
</tbody>
</table>

Module with ‘∗’ denotes change propagation within the module; Module underlined indicates changes found by one or more participants.

The dependency types used in change propagation analysis are listed after modules.

- **Precondition.** In this case, Precondition reflects the business rule or business process between two requirements. For example (Figure 5), in PVS v2.0, after the valuer submits valuation through PDA system, the manager must check the valuation quality on the web system manually. In PVS v3.0, the quality of valuation should be automatically checked on the PDA system before submission. This change will affect the function ‘valuation submission’ on the PDA system and the quality checking function on the web system.

In addition, there are some dependency types propagating changes only at the requirements level. These dependencies are **Purpose, Generalise** in the P-model. Some dependency types do not propagate changes. They are **Example_for, Test_Case_for, Background, Comments** in the P-model and **Increase/Decrease_value_of, Increase/Decrease_cost_of** in the D-model.

### 4.3. Effectiveness of dependency types (RQ3)

Dependence types provide the clue to analyse the change impact. For the case study in particular, their effectiveness includes:
1. Dependency types are helpful to find more dependencies. Participant A reflected that he never knew there are so many dependency types in requirements. When he was analysing the change impact before, he usually identified what requirements were impacted directly only, but ignored those requirements may be impacted indirectly. He thought Constraint and Similar_to were very useful dependency types but new to him.

2. Dependency types support quality analysis of requirements. Participant B reflected that dependencies helped her learn the structure of requirements and relationships in requirements. Through analysing the dependency types in requirements, she found some quality problems in the requirements document.

3. Dependency types are helpful to identify the key dependent functions in the system. Participant C reflected that dependency types helped her in identifying dependent functions. The changes happening in these modules may impact more modules than others. This finding is useful to design and code improvement.

However, we also found ineffectiveness of dependency types in this case study, because some dependency types have ambiguous definitions or have very general semantics (cf. Section 4.1). A few dependency types overlap with each other and are not clearly differentiated. These problems result in the difficulty in dependency identification and change propagation analysis.

4.4. Factors affecting dependency discovery (RQ4)

The number of dependencies found in PVS v2.0 requirements by three participants are listed in Table 6. There exist significant differences among the dependencies found by three participants.

Table 6: Dependencies identified per participant

<table>
<thead>
<tr>
<th>Dependencies found</th>
<th>Participant</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive dependencies found individually</td>
<td>89</td>
<td>69</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Dependencies also found by others</td>
<td>23</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Wrong dependencies identified individually</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>112</td>
<td>82</td>
<td>108</td>
<td></td>
</tr>
</tbody>
</table>

We further interviewed participants to explore the possible reasons for the difference and found four main factors affecting the discovery of dependencies.

(1) Participant’s experience and understanding about the requirements

Individual experiences and understanding about the requirements play an important role in dependency identification. Participant A found most dependencies since he was the project manager and most familiar with the requirements. C also found more than 100 dependencies, but 23 of which are wrong dependencies due to her misunderstanding of some requirements (the wrong dependencies were revealed and confirmed later in interviews and consensus building).

(2) Analysis process and time spent

The task assigned to the three participants was to analyse change propagation in PVS v2.0 requirements using the 25 dependency types from the P-model and the D-model. They all first learned the requirements dependency types, but the ways they completed the task are slightly different.

Participant A started with v3.0 requirements and identified their initial impacts on v2.0 requirements. He then identified dependencies between the initially impacted requirements and the rest of v2.0 requirements. B roughly analysed the dependencies in v2.0 requirements first, then analysed the change propagation and improved the dependencies. While C first identified the dependencies in v2.0 requirements, and then analysed the change propagation.

The time spent by three participants is shown in Table 7. The efficiencies (dependency number by time spent) of A and B are higher than C because the change-driven dependency identification process helped them find dependencies and requirements impacted faster. C found more than 100 dependencies but she spent the most time.

Table 7: Time spent per participant

<table>
<thead>
<tr>
<th>Participant</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change learning</td>
<td>1.5</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Dependency identification</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Change propagation analysis</td>
<td>1.5</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>Total (hr)</td>
<td>3.5</td>
<td>3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

(3) Participant’s perspective

As shown in Table 6, there are differences among the dependencies found by the three participants. We further analysed the dependency types discovered by the three participants (Figure 6). We found a person’s viewpoint is one main factor affecting the dependencies identified.

Participant A identified the dependencies from both business and implementation perspectives. Accordingly, he found many dependencies within the types of Similar/Similar_to, Constraint, and Precondition/Requires. Note that Similar/Similar_to exists between similar user interfaces, Constraint often describes the relationships between functional and nonfunctional requirements, while Precondition/Requires depicts the business process.

Participant B paid more attention to the text representation of the requirements and found many structure and content dependencies such as Comments, Similar/Similar_to, and Contradicts. Different from Similar/Similar_to
found by A, Similar/Similar_to dependency types found by B are mainly similar textual content between multiple requirement descriptions.

Participant C discovered dependencies from a developer’s point of view. C cared more about how to implement the requirements and accordingly she found many Precondition/Requires dependencies. C thought that this kind of relationships often reflects the invocations between code modules.

(4) Representation of requirements documentation

Table 8 shows the dependency distribution within each individual module and between modules. We compared the dependencies found by A, B and C (Figure 7). A found the most dependencies among modules which reflect the business process information across the whole system. This business knowledge is not explicitly expressed in the requirements which are often isolated items and documented in natural languages. B and C only found some process-oriented cross-module business knowledge in the document. It is observed in this case study that natural language based requirements work not very well in capturing business processes, which may limit the identification of dependencies. Multi-representation of requirements (e.g., text and process diagram) can help people effectively find dependencies.

5. A New Requirement Dependency Model

In terms of our evaluation experience from the case study, the biggest difficulty in discovering dependencies in requirements was some ambiguous and overlapping definitions of dependency types. Thereafter, based upon the existing dependency models from the literature (especially the two generic models discussed above [24, 9]) as well as our empirical evaluation, we propose a new dependency model to depict the links between different relationships at the concept level. The new model clearly defines common dependency types to help practitioners identify dependencies among requirements.

5.1. Requirement Dependency Classification

With reference to the categorisations of the two evaluated models (P-model and D-model) and the participants’ difference perspectives reflected in the case study, the new conceptual model proposed in this section classifies dependencies into six categories, i.e. business, implementation, structure, evolution, value, and cost (as shown in Figure 8). On the top level of this model, these dependency categories are further classified into intrinsic dependencies and additional dependencies. Intrinsic dependencies depict essential interrelated state of requirements and reflect semantic and structural information of requirements. This kind of dependency is often related to various software engineering activities and also can be used to analyse change impact [24]. For example, two requirements with Precondition dependency [24] may need to be implemented in one product. Additional dependencies are the relationships existing for specific software engineering activities. For instance, to plan product release, value-related relationships (e.g., Increase_value_of [9]) are defined to discover those requirements necessary for a software product. Product release planning may bring additional dependencies to requirements.

Compared to additional dependencies, intrinsic dependencies are more important for software engineers in discovering dependencies because they may impact many software engineering activities such as change impact analysis and project planning. They are also generally helpful to identify additional dependencies. For example, one additional dependency Increases_cost_of [9] can be determined by one intrinsic dependency Precondition [24].

To identify intrinsic dependencies, we need to analyse the source of these dependencies. Intrinsic dependencies usually come from business knowledge and requirements characteristics. Business knowledge includes business processes and rules which are vital input of software requirements elicitation and analysis. Our previous research [21] found that workflow patterns in a business
Table 8: Dependencies identified within/between modules

<table>
<thead>
<tr>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>M9</th>
<th>M10</th>
<th>M11</th>
<th>M12</th>
<th>M13</th>
<th>M14</th>
<th>M15</th>
<th>M16</th>
<th>M17</th>
<th>M18</th>
<th>M19</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>P(1)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>P(1)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>P(1)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>P(1)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>P(1)</td>
<td>P(2)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
<tr>
<td>S(3)</td>
<td>S(1)</td>
<td>S(1)</td>
<td>C(2)</td>
<td>C(2)</td>
<td>C(1)</td>
<td>R(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>C(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
<td>P(2)</td>
<td>P(1)</td>
</tr>
</tbody>
</table>

Dependency types: P: Precondition, S: Similar to, C: Contradict, R: Refines, B: Based_on, S: Satisfies, C: Comment

Note: the number in parentheses indicates the number of dependencies identified within/between modules.

process model can efficiently help practitioners discover requirement dependencies. These dependencies include temporal and sequential relationships reflected in a business process, such as Requires [7, 9], And, Temporal [7, 25], Task, Causality [25]. These kinds of dependency are ‘business’-related relationships as shown in Figure 8.

Another important source of dependency is characteristics of requirements. These characteristics come from specification structure-related, implementation-related, and evolution-related requirements (as shown in Figure 8).

Different specification structures may produce various dependencies among requirements. One general dependency is Generalised to [9] which means one requirement is explained by another more detailed requirement(s). This type of dependency often indicates a hierarchical structure of requirements. For another instance, if one requirement document template includes background related and example related contents, this creates Example_for and Background [24] dependencies among requirements. This category of dependency are specified such as Refines [24], Generalised_to, Generalise [9] in different dependency models.

Requirement implementation-related dependencies include relationships among functional requirements, non-functional requirements and user interface requirements. For example, some requirements may be the precondition of other requirements [24], and implementation of some requirements may constrain other requirements’ implementation [9]. This category typically includes requirement dependencies like Constraint, Conflicts [24], and
Conflicts_with [9].

Requirement evolution creates relationships between different versions of one requirement. Evolution-related dependency represents requirement change history. This type of relationship includes Replaces, Satisfies, Based_on, Elaborates, Formalises, and Changes_to [24].

Additional dependencies include value and cost categories. Value indicates user’s satisfaction. Value-related dependency types are Increase/Decrease_value_of [9], Positive_value, Negative_value [17], and CValue [7]. The Cost-related dependency types include Increase/Decrease_cost_of [9], Positive_cost, Negative_cost [17], and ICost [7]. These two types of dependency are useful for some specific activities, such as project planning and product release planning.

5.2. Requirement Dependency Types

The case study reflects that dependency discovery is a subjective process. It is impacted by people’s understanding of requirements and relationships to a large extent. To alleviate this subjectivity, the definitions of dependency types need to be very specific, explicit and clear. The more specific the dependency is, the easier for participants to discover this relationship.

The conceptual model (shown in Figure 8) depicts an overall categorisation of requirement dependencies derived from the above discussion on dependency classification. This subsection completes the dependency model with specific but refined dependency types based on the problems identified from Pohl’s model [24] and Dahlstedt’s model [9] in the case study and our previous work [21] as well as the analysis of existing requirement (inter-)dependency types in the literature. We selectively propose and refine nine dependency types (as shown in Table 9) under the categories of the conceptual model at the initial stage. These nine dependency types are defined by 1) removing dependency types not common or seldom found in practise (such as Text_case_for and Purpose [24]), 2) clarifying the dependency types with confusing or ambiguous definitions (e.g., Conflicts [24] and Conflicts_with [9]), 3) combining and refining some overlapping or alternative dependency types in different models (e.g., Similar [24] and Similar_to [9]), and 4) introducing new dependency types (e.g., Be_exception_of).

In addition, to minimise the possible ambiguity, for each dependency type we specify its relationship direction and suggest software engineering activities where the dependency can be used. Such information is not provided in the existing (inter-)dependency models. Table 9 clearly displays the categorisation of the nine dependency types, their sources in existing dependency models, and the differences from other models, in particular the P-model and D-model. The rest of this section describes the definitions with examples of these refined dependency types: 1) Constrain, 2) Precede, 3) Be_similar_to, 4) Refine, 5) Be_exception_of, 6) Conflict, 7) Evolve_into, 8) Increase/Decrease_cost_of, and 9) Increase/Decrease_value_of.

1) Constrain

Definition. One requirement is a constraint of another requirement. This kind of dependency can represent cross-cutting relationship among requirements.

Example. ‘system should respond to users in 3 seconds’ constrains ‘book search function’ (user searches books using book title or ID).


Direction. If A constrains B, the direction is A → B.

Typical use. architecture design, test case design, requirement management, and change propagation analysis.

2) Precede

Definition. If function A precedes function B, A is a precondition of B.

Example. Given Requirement A: ‘a client proposes the valuation request and system checks the quality of request’ and Requirement B: ‘after the quality of request is checked, system allocates the valuation task to a valuer’, then A precedes B.

Category. Business.

Direction. If A is a precondition of B, the direction is A → B.

Typical use. test case design, change propagation analysis.

3) Be_similar_to

Definition 1. If two requirements share similar data information, these two requirements are similar to each other.

Example. Requirement ‘system administrator adds a new book record’ is similar to ‘system administrator modifies an old book record’. Book record in these two requirements is identical.

Definition 2. If two requirements complete similar tasks, these two requirements are similar to each other.

Example. After a client proposes a property valuation request, one valuer can choose from several options to evaluate the property, such as ‘desktop valuation’, ‘curbside valuation’ or ‘full valuation’. Then these options (requirements) are similar to each other.

Category. Implementation.

Direction. If A is similar to B, the direction is A ↔ B.

Typical use. architecture design, change propagation analysis.
Table 9: Comparison and relations of dependency types between the new model and the previous models

<table>
<thead>
<tr>
<th>Relation</th>
<th>Dependency type in new model</th>
<th>Corresponding type in previous models</th>
<th>Difference from previous models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refinement</td>
<td>Constraining</td>
<td>Constraining [24]</td>
<td>Extension</td>
</tr>
<tr>
<td></td>
<td>Precede</td>
<td>Precondition [24], Requires [9]</td>
<td>Extension</td>
</tr>
<tr>
<td></td>
<td>Be_similar_to</td>
<td>Similar [24], Similar_to [9]</td>
<td>Extension</td>
</tr>
<tr>
<td>Conflict</td>
<td></td>
<td>Conflicts [24], Conflicts_with [9]</td>
<td>Clearer definition with equivalent meaning</td>
</tr>
<tr>
<td>Increase/Decrease_value_of</td>
<td>Increase/Decrease_value_of [9]</td>
<td>Clearer definition with equivalent meaning</td>
<td></td>
</tr>
<tr>
<td>Increase/Decrease_cost_of</td>
<td>Increase/Decrease_cost_of [9]</td>
<td>Clearer definition with equivalent meaning</td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>Be_exception_of</td>
<td>Combine, Replaces, Satisfies, Based_on, Formalises, Elaborates, Changes_to</td>
<td>New dependency type</td>
</tr>
<tr>
<td>Evolve_into</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heritage</td>
<td>Refine</td>
<td>Refines [24], Refines_to [9]</td>
<td>Definition from previous models</td>
</tr>
</tbody>
</table>

(4) **Refine**

Definition. One requirement is refined by more specific requirements.

Example. ‘A valuer submit a valuation report’ is refined as ‘a valuer submits a report through web system’, ‘a valuer submits a report through PDA’.

Category. Structure.

Direction. If A is refined by B, the direction is B → A.

(5) **Be_exception_of**

Definition. One requirement describes the exceptional event of another requirement.

Example. “A user inputs a null username and system reminds him to input his correct username” is an exceptional event of “a user inputs his username and password and system checks validity of his information”.


Direction. If A is an exceptional event to B, the direction is A → B.

Typical use. test case design.

(6) **Conflict**

Definition. Implementation of one requirement negatively impacts another requirement.

Example. Increasing security might negatively impact system performance.

Category. Implementation.

Direction. If A conflicts with B, the direction is A → B.

Typical use. architecture design.

(7) **Evolve_into**

Definition. If one requirement B is a new version of another requirement A, then A evolves into B. This type of dependency is used to trace and compare different versions of requirement documents.

Example. “Clients use a desktop PC to upload reports” evolves into “clients use PDA to upload reports”.

Category. Evolution.

Direction. If A evolves into B, the direction is A → B.

Typical use. requirements tracing.

(8) **Increase/Decrease_cost_of**

Definition. The implementation of one requirement causes the increase/decrease of the implementation cost of another requirement.

Example. If one requirement states that ‘no response time should be longer than 5 seconds’, it is most likely to increase the cost of implementing many other requirements.

Category. Cost.

Direction. If A increases/decreases cost of B, the direction is A → B.

Typical use. product releasing planning, project planning.

(9) **Increase/Decrease_value_of**

Definition. The implementation of one requirement causes the increase/decrease of the value to the customer of another requirement. Note that ‘value’ here mainly reflects customer’s satisfaction.

Example. Users can listen to MP3 when they are browsing photos in a mobile phone. In this case, ‘users can listen to mp3’ increases customer’s satisfaction of ‘users can browse photos’.

Category. Value.

Direction. If A increases/decreases value of B, the direction is A → B.

Typical use. product releasing planning, project planning.
6. Discussion

6.1. Dependency Types for Change Propagation

Based upon the empirical findings from the case study, we propose a group of dependency types which are particularly useful in change propagation analysis. We redefine the dependency types’ definitions and provide the specific change patterns. These dependency types can help software engineers identify changes and they also can be integrated with the requirements modelling languages to aid change analysis in requirements models. We will validate these dependency types in our future study. We classified the dependency types into two categories:

(1) Dependency types for direct propagation

In the new dependency model, Evolve_into is a generic dependency type for direct change propagation. This dependency type can be further specified as several dependency types (cf. Table 9) specific to more concrete scenarios, which describe the impact extent of the new requirements on the old requirements.

- **Elaborate**: The new requirements are the refinement of the old requirements and might be the subcontents of the old requirements. This kind of change may impact the old requirements’ implementation and result in the modification in requirements documents.

- **Replace**: The new requirements replace the old requirements. The old requirements may be discarded. This kind of change may cause the code refactoring and the modification in requirements documents.

- **Combine**: The new requirement and the old requirement are combined into one integral requirement. This kind of change may cause the modification in requirements document and the old requirement’s code refactoring.

(2) Dependency types for indirect propagation

Some dependency types in the existing dependency models (e.g., P-model and D-model) reflect the structure and the content relationships in requirements such as Test_case_for and Example_for. These types only propagate changes at the requirements level and are not included in our dependency type sets. Here we only redefine the dependency types that may cause changes at the implementation level and provide the change patterns.

- **Be_similar_to**: The data information of one requirement overlaps with another requirement.

  Change Pattern: Given A is similar to B, when the data information of A is changed, B’s implementation may be changed as well. For example, when the data information is changed, user interfaces of functions dealing with all this data information are very likely to be impacted.

- **Precede (Precondition)**: Only after one function is completed or one condition is satisfied, another function can be performed. Usually precondition reflects the business rule or the logical/sequential relationship between tasks.

  Change Pattern: Given A precedes B, when A is changed, B’s implementation is likely to be changed. When the business rule or business process is changed, the related requirements’ implementation may also be changed.

- **Constrain**: The relationships exist between the non-functional requirements and the functional requirements or between the functional requirements. The crosscutting relationship in aspect-oriented requirements engineering is one kind of this dependency.

  Change Pattern: Given A constrains B, A’s change may lead to the change of B’s implementation; and when B is changed, A’s implementation may be changed too. To avoid change propagation in a large scope, in many cases, A needs to be implemented in an individual module instead of being embedded in many modules.

6.2. Limitations of the Case Study

The case study reported in this article is the initial empirical evaluation of the applicability of requirements dependencies in the community, which may have two major limitations at this stage:

(1) Small sample size

The group size participating in the case study is relatively small. Only three people participated in this case study. However, these three participants took representative roles and the case setting is in a real industry project. They possess different backgrounds and experiences such as project management, design, requirements analysis or development. Their findings help us learn that people with different backgrounds may have different viewpoints about change propagation analysis. In addition, two participants are co-authors of this paper. They strictly followed the procedure in this study. Furthermore, the original goals of the exercise were to simply use the dependency types for identification and change analysis in a real project and conduct an initial evaluation with an exploratory and experience report nature. At this stage, there were no strong hypotheses involved.

(2) Single project

For external validity, we have only evaluated dependency types on one project as our first step towards the evaluation and improvement of our new dependency model and acknowledge its limitation. Although we only conducted the case study in one project that might to some extent impact the participants’ findings, this application appeared to be fairly complex and the project is representa-
tive in real development settings. The system, PVS, comprises three subsystems: web subsystem, desktop subsystem and mobile (PDA) subsystem being connected through networks. This system manipulates and manages a great amount of data across subsystems and organisations, and has high security and performance requirements. In addition, the business process in this system is complex. There are more than one hundred requirements. The current goal was to conduct an initial in-depth case analysis for more systematic evaluation in the near future.

The case study discussed here is the first real-world applicability evaluation of dependency types in both identification and change propagation analysis. We acknowledge that the new dependency model and the associated dependency types proposed in this article are mainly based on the existing literature and the findings of this case study which needs to be validated. Compared to other existing dependency models, however, we formalise the definitions of the initial set of dependency types with their explicit direction and typical use in software engineering. We believe the refined definition of the dependency types can facilitate the realisation of other empirical studies similar to ours.

7. Conclusions and Future Work

Although the existing requirement dependency literature suggested many dependency models, there still lacks an overall comprehensive understanding about dependencies among requirements. This article reports our research in evaluating and refining the dependencies inherent in software requirements. We first conducted a case study to empirically evaluate the 25 dependency types from two generic requirement dependency models in an industry project. As the initial evaluation of the dependency models/types in the community, this study focuses on evaluating the usefulness and applicability of these dependency types in identification and change propagation analysis. From the analysis of the results, our findings are 1) the definitions of some dependency types are confusing and difficult to use in practise; 2) some dependency types from literature are not common and seldom found in real projects; 3) in existing dependency models, five dependency types propagate changes, but their definitions need to be clarified.

We also found change propagation analysis is affected by a practitioner’s viewpoint and experiences. The participant with project management experiences may emphasise the changes that happen at the business and high-level design level. The participant with requirements engineering experiences pays special attention on the changes at the level of requirements description and business. While, the participant with development experiences may care more about the changes happening at the implementation level. All their concerns are important for change analysis. Therefore, change propagation analysis should involve a wide range of stakeholders.

Based on these findings from the case study as well as our previous research in evaluation and identification of requirement dependencies, we propose a new requirement dependency model that aims to depict a clear classification of requirement dependencies and their interactions with software engineering activities. Nine specific dependency types are suggested in the initial version of the new dependency model which have refined, improved and integrated existing requirement dependencies. We expect other researchers’ suggestions and additions to further improve this dependency model.

Dependency identification is also useful to key requirements elicitation. Our initial case study discovers and provides a group of dependency types (with change patterns) important to change propagation analysis. These dependency types can be used not only in the natural language requirements documents, but also in the requirements models defined by requirements modelling languages. In the future, we will apply our findings to more industry projects to validate their applicability.

Future work is planned in various directions. First, the definitions of dependency types in the new model need to be improved. Although the model is based on the existing literature and the empirical evaluation of other dependency models in one industry project, more examples are still needed to increase their understandability. Currently we are undertaking follow-up empirical evaluation of the new requirement dependency model proposed in this article. The follow-up studies focus on the understandability and usability of the new model in comparison with the conventional P-model and D-model as well as serve as the replication of the initial study reported here but with improved internal and external validity, e.g., more rigorous design, larger sample size, and more projects from different countries. The details and results of the follow-up studies will be reported in the near future.

Next, the representation or modelling method of requirement dependencies needs to be explored. A requirement dependency modelling language can represent different types, directions and contents of relationships as well as model the evolution state of relationships during the whole product lifecycle.

Last but not least, we encourage software engineering researchers carry out more empirical studies to evaluate the existing dependency models, and particularly welcome the external evaluation of the new dependency model suggested in this paper.

Acknowledgments

This work is supported by National ICT Australia which is funded through the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council. This work is also supported by the National Natural Science Foundation of China under grant No. 60803023, No. 60873072, No. 90718042, the National Basic Research Program (973 program) under
grant No. 2007CB310802, the Hi-Tech Research and Development Program of China (863 Program) under grant No. 2007AA101030

References


A. Dahlstedt and Persson’s Interdependency Model

Dahlstedt and Persson proposed a requirement (inter-)dependency model [9]. This section provides a concise overview of the seven (inter-)dependency types suggested in the D-model (Figure 2). More details can be found in [9].

A.1. Structural Interdependencies

Structural (inter-)dependency types are organised in a structure where relationships are of a hierarchical as well as of a cross-structure nature. The following (inter-)dependency types fall into this category.

- **Refined_to**. A higher-level requirement is refined by a number of more specific requirements. It is used to describe hierarchical structures, where more detailed requirements are related to their source requirements. In this sense, these requirements provide further explanation, detail or clarification about the source requirement. The source requirement can hence be seen as an abstraction of the detailed requirements.

- **Changes_to**. One requirement changes to another requirement if a new version of that requirement is developed which replaces the old one. It is used to describe the history of a requirement, i.e., how it has evolved over time since it enables to relate the different versions of a single requirement.

- **Similar_to**. One stated requirement is similar to or overlapping with one or more other requirements. It describes situations where one requirement is similar to or overlapping with another in terms of how it is expressed or in terms of a similar underlying idea of what the system should be able to perform. It can also be used to describe similar solutions, from which one has to be selected to be part of the system.

A.2. Constraining Interdependencies

These (inter)dependencies can be identified in order to describe how requirements can constrain each other or be dependent on each other, especially if this classification is further elaborated with respect to different development activities or decisions. There exist two types within this category.

- **Requires**. The fulfilment of one requirement depends on the fulfilment of another requirement. It is used to describe that if one requirement is to be included into the system, it requires another requirement to be included as well. It can also be used to describe hierarchical relations between two requirements of a stronger nature than refined_to, i.e. not optional.

- **Conflicts_with**. A requirement is in conflict with another requirement if they cannot exist at the same time or if increasing the satisfaction of one requirement decreases the satisfaction of another requirement. It both includes situations were it is impossible to implement both requirements, and situations where requirements have a negative influence on each other’s achievement and a trade-off between the resolution of the requirements must be made.

A.3. Cost/Value Interdependencies

These (inter-)dependencies are concerned with the costs involved in implementing a requirement in relation to the value that the fulfilment of that requirement will provide to the perceived customer/user.

- **Increase/Decrease_cost_of**. If one requirement is chosen for implementation, then the cost of implementing another requirement increases or decreases. It is used to relate requirements that somewhat influence the implementation, cost of each other.

- **Increase/Decrease_value_of**. If one requirement is chosen for implementation, then the value to the customer of another requirement increases or decreases. It focuses on the effect relations between requirements may have on the perceived customer value.

B. Pohl’s Dependency Model

Based on a literature survey in the area of requirements engineering, Pohl proposed a requirement dependency model [24] that includes 18 dependency types which are grouped into five categories (shown in Figure 1: condition, content, documents, evolutionary, and abstraction. For example, document-related dependency types are embedded in the structure, content and version relationships in the requirements representation. Pohl’s model introduces broad and general interdependency types that can exist between any type of trace object used in the requirements engineering process. To keep it brief, we selectively elaborate a few common dependency types below. More detailed description and explanation can be found in [24].

- **Constraints**. A requirement can relate to another by being a constraint to the latter. For instance, the requirement “cash withdrawal is limited to $2000 daily” is a constraint for the requirement “withdraw cash”.

References


• **Refines.** One requirement can be a refinement of another requirement, providing more detailed descriptions for the latter.

• **Precondition.** Only after one function prescribed by a requirement is finished or one condition described by it is satisfied that another function prescribed by a requirement can be performed. Usually *Precondition* reflects the business rule or the sequence relationship between tasks. For instance, the requirement “a customer successfully logged in to the system” is a precondition for “a customer withdraws cash from bank account”.

• **Satisfies.** This type expresses that if one requirement is implemented in the system, another requirement is also satisfied in it. For example, the requirement “when a user quits, all opened documents will be automatically saved” satisfies the requirement “no unsaved work will be allowed before the editor terminates”.