RC25081 (C1012-004) December 6, 2010
Computer Science

IBM Research Report

A Collaborative Requirement Elicitation Technique for SaaS Applications

Xin Zhou[1], Li Yi[2], Wei Zhang[2], Ying Liu[1]

[1]IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100193
P.R.China

[2]Peking University
Beijing
P.R. China

A Collaborative Requirement Elicitation Technique for SaaS Applications

Xin Zhou
IBM Research - China
Beijing, China
zhouxin@cn.ibm.com

Li Yi, Wei Zhang
Peking University
Beijing, China
{yili07, zhangw}@sei.pku.edu.cn

Ying Liu
IBM Research - China
Beijing, China
aliceliu@cn.ibm.com

ABSTRACT

Software as a Service (SaaS) provides a web based software delivery model to serve a large number of clients with one single application instance. One of the essential problems to SaaS application development is about how to elicit the commonality and variance of multiple clients' requirements effectively. This paper presents a collaborative requirement elicitation technique (CRETE), which keeps each potential client of a SaaS application aware of the requirements raised by other clients or the SaaS vendor and allows a client to vote on existing requirements or raise new requirements. With CRETE, individual client can create and evolve his proprietary requirements model, while the SaaS vendor can automatically get a combined requirements model that reflects all clients' common and variant requirements. The SaaS vendor then can develop a SaaS application according to the combined requirements model, so that individual client's requirements can be satisfied by self-serve configuration without changing the SaaS application's source code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types*

General Terms

Management

Keywords

SaaS, requirement elicitation, feature model, collaboration

1. INTRODUCTION

Software as a Service (SaaS) provides a web based software delivery model to serve a large number of clients with one single application instance, which has gotten rapidly growing acceptance by software vendors [5][6]. Each client might present variant requirements on the application due to their unique business and/or operational needs. To deal with such variant requirements, SaaS vendors should provide an application with all functionalities and offer clients with configuration and/or customization capabilities to tailor the whole application to a unique one as wanted **Error! Reference source not found.**[11]. If clients' variable requirements can be clearly identified and well dealt with at SaaS application development time, configuration can be easily done at SaaS runtime to meet each client's unique requirements. For those unique requirements not considered at SaaS development time, configuration doesn't work and only customization can be performed with significant complexity and cost.

There are many literatures reporting research works on SaaS application configuration and customization [11][12][14][9]. However, little work is reported on the elicitation of variant requirements for a SaaS application, which is fundamental to SaaS configuration and customization. Without well elicited and organized common and/or variant requirements from potential clients, it's difficult to pre-define enough and appropriate configuration capability for a SaaS application at development time **Error! Reference source not found.**[12]. As the volume of SaaS clients is usually large and the clients are separate, existing traditional requirement elicitation methods [1][3][8] are incapable of collecting large amount of diverse requirements and identifying the commonality and variability among these requirements.

In this paper, we propose a Collaborative Requirement Elicitation Technique (CRETE) for SaaS application development. The basic idea is to keep each potential client of a SaaS application aware of the requirements raised by others and allow them to raise new requirements or vote on existing requirements. With CRETE, individual client can create and evolve his proprietary requirement model, while the SaaS vendor can automatically get a combined requirement model that reflects all clients' common and variant requirements. The remaining part of this paper is organized as follows. Section 2 presents the concept framework of CRETE. The requirement elicitation process is illustrated in Section 3. Section 4 introduces the preliminary experiment conducted to validate the feasibility of CRETE. Section 5 introduces some related works. Section 6 concludes this paper and discusses future works.

2. CRETE Conceptual Framework

2.1 Overview

The purpose of CRETE is to facilitate the vendor and potential clients on collaboratively presenting, verifying and refining a SaaS application's requirements via web. A SaaS vendor can raise an initial set of features and then inform the potential clients to verify those features by voting "yes" or "no" on them. Also, each client can present their personal requirements on the application by creating new features into the CRETE environment, which further triggers other clients to vote "yes" or "no" on these newly-added features. During the whole process, the SaaS vendor can review all requirements about the application.

The conceptual framework for CRETE is shown in Figure 1, which will be further illustrated in following sections.

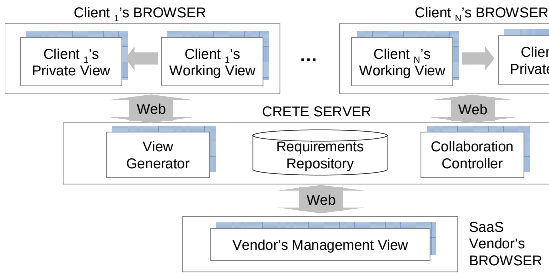

Figure 1. CRETE Conceptual Framework

2.2 CRETE Requirement Repository

CRETE requirement repository stores not only all requirements presented by every clients, but also collaboration-related information (for example, the creation or voting actions a client performs on a requirement).

Requirements information includes "feature", "refinement", and "constraint". Also, the creator of those requirement elements is recorded in the repository. The refinement relationships organize features with different levels of abstraction or granularities into a hierarchical structure. The constraint relationships describe dependencies between features.

Collaborative requirement modeling information includes "direct voting", "propagated voting" and "preference". A direct vote is a "yes" or "no" directly given by a client on an existing element. A propagated voting is automatically applied on an element as a consequence of a directly voting initiated by a client. For example, if there is a direct voting "yes" applied to a refinement relationship, there will be a corresponding propagated voting "yes" to the two features involved in this relationship. We also record a client's preference on an element he votes "yes", include the element's alias, comment to this element's original name or description, etc.

2.3 Client's Views

Client's working view (CWV) is a client's main workspace to build requirement model for a SaaS application. In the CWV, a client can create requirements and echo existing requirements presented by the SaaS vendor or other clients. The entities in the CWV are those "feature" and "refinement" elements that the clients hasn't vote "no". In CWV, "constraint" elements will not be presented for clients to voting. They will only be used at the backend to check the validity of a client's voting.

Client's private view (CPV) is to represent a client's proprietary perspective on the requirements on the SaaS application. The entities in the CPV are those "feature" and "refinement" elements that the client has voted "yes". For an element created by the client, CRETE will automatically add a "yes" voting to the element for this client.

2.4 SaaS Vendor's Views

The SaaS vendor works with his management view (VMV) during the process of collaborative requirement elicitation. In VMV, the vendor can create some initial requirements for a to-be developed application, with which clients can vote on them or be inspired to propose new requirement. Also, the vendor can review all the requirements about an application. Besides, the vendor is presented with each requirement's creator, the supporters and the dissenters. Then, he can understand which requirements are required by which clients. Also, he can attach the requirement relationships on the management view from application development perspective. For example, the "excludes" constraint can be added between two features to indicate their conflict due to development considerations.

3. Collaborative Requirement Construction Process

3.1 Process Overview

The collaboration process for constructing a SaaS application's requirements model is shown in Figure 2, which includes human activities performed by clients and system activities performed automatically by the system according to predefined rules.

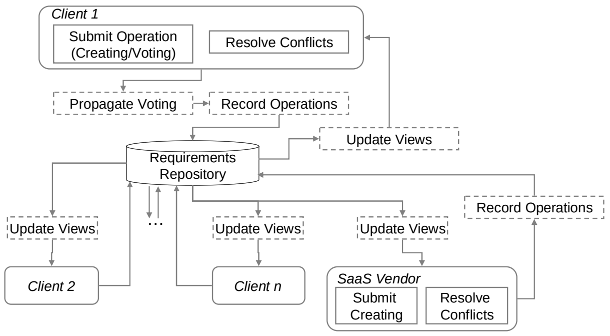

Figure 2. Requirement Construction Process

Update Views: When the information in the requirement repository changes due to operation submission or propagated voting, client and vendor's views can be automatically updated.

Resolve Conflicts: When a client's working view is updated or he submits an operation, he should first focus on the conflicts brought to the view and resolve them using creating and/or voting operations. Conflict resolving will be elaborated in Section 3.2.

Submitting Operations: In addition to conflict resolution, a client constructs his requirements by submitting operations. He creates a requirement totally new by submitting creating operation and adds a requirement already presented by others by submitting "yes" voting operation. To remove those requirements already presented by others from his working view, a client should submit "no" voting. For SaaS vendor, he can create initial requirements model for clients' reference or add relationship to his management view by submitting creating operations.

Propagate Votes: Propagated votes are computed according to the rules in Table 1 after an operation was submitted:

Table 1. Voting propagation rules

ID	Rules
PR-1	Vote "*yes*" on Refinement r -> Vote "*yes*" on Feature f, f is involved in r
PR-2	Vote "*no*" on Feature f -> Vote "*no*" on Refinement r, f is involved in r

PR-3	If there is a "f1 *excludes* f2" constraint, Vote "*yes*" on f1 -> Vote "*no*" on f2, Vote "*yes*" on f2 -> Vote "*no*" on f1
PR-4	If there is "f1 *requires* f2" constraint, Vote "*yes*" on f1 -> Vote "*yes*" on f2

Coordinate Operations: In the context of collaborative work, changes submitted by multiple clients need to be coordinated. After the coordination, if the original changes are still valid, they are stored in the CRETE requirement repository and in turn, cause the update of views of all clients; otherwise the changes are neglected and its submitter will be informed. Operation coordination will be elaborated in Section 3.3.

3.2 Conflict Resolution

Conflicts might exist in a client's working view as the requirement elements presented in it are from multiple clients. Meanwhile, a client's operations may also bring conflicts to his working view. The potential conflicts in a client's working view include:

Non-positioned Feature (NPF): the client has denied all existing positions of a feature without giving a new one. (A feature must be positioned as either a root feature, or a child of another feature.)

Conflicting Refinements (CR): multiple refinements existing in a working view are considered conflicting if they involve the same feature as the child but different features as the parent.

For a NPF, a client should create a refinement involving it as a child, or make it a root feature explicitly, or reconsider existing refinements. For CR, a client should vote on them to select only one, or even deny all and create a new one.

3.3 Operation Coordination

As different clients might work on the same requirements set while submitting operations, it is possible that their operations need to be coordinated. According to the operation types (creating or voting) that cause the repository updates, three possible situations to be coordinated are listed as below.

Duplicate creation happens when a client (c2) creates an element e1 before a previous creation of the same element has became visible (i.e. update c2's working view) to him.

Unreachable vote is a vote on a nonexistent element. If client c1's "no" vote on element e1 leads to the deletion of e1, and if client c2 submits a "yes" vote on e1 before the deletion becomes visible to him, then c2's "yes" vote is unreachable.

Unreachable propagation is similar to unreachable votes. If client c1's "no" vote on feature f1 leads to the deletion of f1, and if client c2 creates a constraint involving f1 before the deletion becomes visible to him, then the propagation of "yes" vote on f1 (according to the rule PR-1) is unreachable.

Coordination of these situations follows a serialized update strategy, that is, all update applies to the elements in the same order of their submission. For duplicate creations, the first creating operation adds a new element to the repository, and the latter is converted to a "yes" vote. For unreachable votes, they are no longer valid on a nonexistent element and are neglected. For unreachable propagations, they are neglected together with the operations which cause the propagations.

4. An Experiment

We conduct an experiment with five participants acting as four clients and one SaaS vendor, respectively. The scenario we designed for them is to collaboratively eliciting requirements for a SaaS application that supports multiple enterprises to register on it as individual tenant to perform on-line recruiting related activities like position publishing, position applying, interview arranging, etc.

To facilitate the experiment, we develop a tool that implements all essential aspects of CRETE, including automatic generation of client and vendor views, support of creating and voting on features, and organizing features with refinement relationships. The five participants spend 3 hours working simultaneously in the first day, and spend a couple of hours working freely (at any time they like, and often at different time) in the next two days. In the end, all clients confirm the requirements in their private views, and the vendor gets the requested features for the system in his management view. There are totally 113 features proposed for the application, including 30 variant features. Two interesting observations have emerged from this case study:

One observation is that the efficiency of requirements elicitation is greatly improved. The reason is that participants are often inspired by others' work. Table 2 shows the proportion of feature creation and reuse in each client's private view, which is another evidence for the improvement of efficiency of our requirements elicitation.

Table 2. Features in each client's private view (CPV)

Client	Total number of features in CPV	Number of features in CPV that are created by the client	Number of features in CPV that are voted "*yes*" by the client
Client 1	91	21 (23.1%)	70 (76.9%)
Client 2	87	37 (42.5%)	50 (57.5%)
Client 3	94	34 (36.2%)	60 (63.8%)
Client 4	104	21 (20.2%)	83 (79.8%)

The other observation is that many variants can be observed in the vendor's management view, as shown in Table 3. It implies that our approach is capable of capturing variant requirements among the clients. The "common structure" emerges by comparing the structure of each client's private view. We have observed that all conflicting are resolved by the participants finally, and all private views have the similar hierarchical structure except that some of the leaf features are different.

Table 3. Common and variant features in VMV

Total number of features	113
Number of common features in 4 CPVs	83 (73.5%)
Number of features presented in 3 CPVs	24 (21.2%)
Number of features presented in 2 CPVs	5 (4.4%)
113 Number of unique features in 1 CPV	1 (0.9%)

The results preliminarily demonstrate that our approach is suitable for requirements elicitation of SaaS applications, and the explicit support for collaboration between clients and vendors has very positive influence on the efficiency of elicitation.

5. Related Works

5.1 Feature-oriented Requirements Modeling

Our work in this paper is partly inspired by the research of feature-oriented requirements modeling [7][10][13]. One implicit assumption of these works is that there is an available set of domain experts who possess a comprehensive understanding of the current software product line and thus can discover all the important commonality and variability in the software requirements. However, this assumption is generally not hold in the SaaS circumstance, because of the rapid evolution nature of SaaS applications and the low possibility of getting a suitable set of domain experts for SaaS applications.

Our approach releases the feature-oriented requirements modeling from above assumption by integrating it with explicit collaboration mechanism and encapsulating it as a web-based application. The larger number of geographic-distributed clients can express, share requirements in a collaborative and asynchronous way, and the requirement commonality and variability can be collected via analyzing voting on requirements. Furthermore, we proposes an effective evolution mechanism, that is, clients can continuously express their updated requirements about a SaaS application and the SaaS vendor can always get the latest commonality and variability requirements of a large number of clients.

5.2 Collaborative Requirements Engineering

Another important research area related to our work is the research on collaborative requirements engineering. Potts et al. proposed an approach to carry out the requirements elicitation through the iteration of three activities: requirements documentation, discussion and evolution [4]. In CREWS project, the concept of scenario is used to facilitate the conduction of requirements engineering activities in collaborative ways [3]. Decker et al. leveraged wiki to support asynchronous collaborative requirements collecting **Error! Reference source not found.**[2].

All these research works aim to get a suitable set of requirements for only a single customer or organization, and it is almost impossible for them to collect and analyze requirements containing variability requirements from different customers or organizations. While in our approach, a carefully designed voting mechanism is provided to collect requirements from a large number of different customers and to reflect the commonality and variability of those collected requirements, which makes our approach suitable for SaaS applications' requirements elicitation.

6. Conclusions

This paper presents a Collaborative Requirement Elicitation Technique (CRETE) to facilitate SaaS vendors on eliciting the commonality and variance of multiple clients' requirements effectively. In this approach, a center repository is used to record all requirements about the to-be developed SaaS application and all collaboration related information. Based on the information in repository, client's working view is presented as a client's workspace while client's private view is presented for a client to review his proprietary requirement model. SaaS vendor raises initial application requirements and reviews the combined requirements model through management view. A process with relevant conflict resolving and coordination rules is also proposed to guide the collaboration. An experiment is conducted to show the feasibility of CRETE.

In the future, we are going to focus on improving the usability of the CRETE. Especially, how to represent a large number of requirement elements in client's working view in a well organized way so that clients will not get lost in it.

7. REFERENCES

[1] A. Van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective," International Conference on Software Engineering, Los Alamitos, California: IEEE Computer Society Press, 2000, pp. 5-19.

[2] B. Decker, E. Ras, J. Rech, P. Jaubert, and M. Rieth. Wiki-Based Stakeholder Participation in Requirements Engineering. IEEE Software, 2007, 24(2):28-35.

[3] CREWS Homepage. http://sunsite.informati k.rwth-aachen.de/CREWS/.

[4] C. Potts, K. Takahashi, A.I. Anton. Inquiry-Based Requirements Analysis. IEEE Software, 1994, 11(2):21-32

[5] D. Ma, The Business Model of "Software-As-A-Service". In 2007 IEEE International Conference on Services Computing (SCC 2007), pages: 701–702, July 2007.

[6] L. Herbert, E. G. Brown, and S. Galvin, "Competing in the fast-growing SaaS market", Forrester Report, No. 0,5110,44254,00, 2008.

[7] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Architecture", Annals of Software Engineering, vol. 5, pp. 143-168, Sept. 1998

[8] K. E. Wiegers, Software Requirements, Microsoft Press, 1999.

[9] K. Zhang, X. Zhang, W. Sun, H.Q. Liang, Y. Huang, L. Zeng and X. Z. Liu, "A Policy-Driven Approach for Software-as-Services Customization", IEEE Computer Society, CEC-EEE, 2007, pp.1-8.

[10] P. Clements, and L. Northrop, "Software Product Lines: Practices and Patterns", Addison-Wesley Professional, 2001

[11] R. Mietzner, A. Metzger, F. Leymann, K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications", 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pages: 18-25.

[12] W. Sun, X. Zhang, C. Guo, P. Sun, and H. Su, "Software as a Service: Configuration and Customization Perspectives". SERVICES-2. IEEE, 2008, pages: 18–25.

[13] W. Zhang, H. Mei, H. Zhao, "A feature-oriented approach to modeling requirements dependencies," in Proc. of the 13th

IEEE Intl. Conf. on Requirements Engineering (RE 05), 2005, pp. 273–282

[14] Y. Shi, S. Luan, Q. Li, H. Wang, “A Multi-tenant Oriented Business Process Customization System”, 2009 International Conference on New Trends in Information and Service Science, Beijing, China, 2009.