Requirements Reuse and Feature Interaction Management

Article

3 authors, including:

Mohamed Shehata
King Khalid University
22 PUBLICATIONS   **880** CITATIONS

Armin Eberlein
University of Regina
128 PUBLICATIONS   **2,514** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Empirical Study on the Growth of Open Source and Commercial Software Products View project

Project   Knowledge Acquisition for Knowledge-Based Systems View project

Requirements Reuse and Feature Interaction Management

Mohamed S. Shehata   Armin Eberlein
Department of Electrical & Computer Engineering
University of Calgary
2500 University Drive NW
Calgary, AB, Canada
{Shehata; Eberlein}@enel.ucalgary.ca

H. James Hoover
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
Hoover@cs.ualberta.ca

The reuse of artifacts of past software development efforts in future ones is a key issue in software engineering. Because of the nature of software, the opportunities for reuse are richer than most other engineering domains. We regularly reuse portions of code implementations, designs, tests, and in a few cases also requirements. Currently the most common kind of reuse is via components and frameworks, which embody design, implementation and the documentation for how to use the framework. Frameworks, especially domain-specific ones, provide functionality that is common to a set of applications. Thus the user of a framework is also implicitly reusing the common requirements that led to the framework.

Developers realize that complex applications are often best built by using a number of different components, each performing a specialized set of services. But the components, each embodying different requirements in different service domains, can interact in unpredictable ways. How to design components to minimize or at least manage interaction is a current issue. This problem of interaction becomes even more significant when reusing requirements. Interactions must be detected and resolved in the absence of a specific implementation framework.

The understanding of interaction management is key to understanding how to reuse requirements. This paper introduces a conceptual process framework for formulating and reusing requirements. We classify reusable requirements into three different levels of abstraction for software requirements. We use this classification in a reusability plan to support our view of the importance of interaction management. The new reusability plan includes the requirements engineering phase in addition to interaction management as a third axis.

Keywords: Requirements engineering, reuse, requirements interaction management.

1. Introduction

Software reuse is the process of reusing existing software artifacts in building a new system rather than starting development totally from scratch. The development of software systems based on reusable components has been present from the beginning of computing. What has changed over time is the size and complexity of the artifacts being reused. The arguments for reuse remain the same: we can do more, with higher quality, for lower cost. Because products are getting more complex with shorter times to market, there is pressure to reuse larger artifacts.

Reusability can be introduced at different phases in the software development life cycle, from requirements to design, to deployment; and at different granularity from subroutine libraries to frameworks to application servers. For example, at small granularities excellent results have been achieved with the reuse of source code in the form of libraries, and with designs in the form of algorithms and design patterns. The higher the level of abstraction at which reuse takes place, the greater the benefit, as is demonstrated by the extensive use of frameworks in contemporary development. The largest granularity of reuse incorporates a very high level of abstraction such as the integration of complete products into a complex system.

Most of the research in the area of reuse neglects requirements engineering, except for the kind of commonality and variability analysis which goes on to develop a framework after the requirements for the product line have been established. Although it is argued [1] that requirements reuse can introduce even more reusability at later stages in the product life cycle, it was not until recently that requirements reuse has received greater attention by researchers.

Most requirements reuse is done in an informal manner by developers who can develop new requirements specifications quicker because they have developed similar products before. Their experience helps them to informally reuse requirements. The fact that there is so much in common between many applications in a domain is what led to the development of frameworks, which embody design, implementation and the

documentation for how to use the framework. Frameworks, especially domain-specific ones, provide functionality that is common to a set of applications. Thus the user of a framework is also implicitly reusing the common requirements that led to the framework. But these common requirements are usually the result of examining the individual requirements of a number of products, not from any systematic effort at requirements reuse. That said, frameworks are still the most common form of achieving requirements reuse at present.

Framework practitioners realize that complex applications are often best built by using a number of different frameworks, each performing a specialized set of services [2]. But the frameworks, each embodying different requirements in different service domains, can interact in unpredictable ways. How to design subframeworks to minimize or at least detect and resolve interactions is a current issue in the framework community.

This interaction problem is even more significant at the requirements level. At least with frameworks there is an implementation to analyze. Much of the interaction analysis that could be done at requirements time is moved to framework specialization time. When dealing with requirements alone we have to consider many possible choices for implementation (even if we use frameworks), and it is not all clear what we need to say in our reusable requirements so that we can even articulate potential interactions, never mind resolve them. It is our conviction that requirements reuse must be done in conjunction with a careful management of the relationships and dependencies that arise between reused and new requirements.

We define requirements reuse as the process of analyzing, eliciting and managing requirements at a suitable abstraction level so that they can be reused in new systems. Requirements Interaction Management (RIM) was first introduced by W. Robinson [3] as "the set of activities directed towards the discovery, management, and disposition of critical relationships among a set of requirements". One might argue that interaction analysis is only necessary at the time when the reusable requirements are created, and since these reusable requirements will not change, no further interaction analysis is necessary when a new system is built based on the existing reusable library. However, one has to consider that there are parameterized requirements that need specific values, that is, they are really "new" requirements. Therefore the big question is how to ensure that the values that are assigned to the parameterized requirements or the new requirements that are added to the specification will not interact with the already existing requirements? This can only be ensured by continuously analyzing the requirements for any interaction that might occur. Furthermore, requirements evolve over time, so the process of interaction management must be continuous.

The first part of this paper deals with introducing reusability at the requirements level and presents a general process framework for eliciting and formulating requirements in a reusable format based on the current research that has been conducted by different institutes and research organizations. The development of this process framework helped us in two ways: We gained a better understanding of how requirements can be reused. The second benefit, which we believe to be of great importance, was the identification of the different levels of abstraction for software requirements. In the second part of the paper, this classification is used to introduce our proposed reusability plan, which is an extension of the software reusability plan familiar to framework developers.

The rest of this paper is organized as follows: Section 2 presents the importance of requirements interaction management. Section 3 presents our understanding of requirements reuse in the form of a general process framework. Section 4 combines our view of the importance of requirements interaction management with requirements reuse through the introduction of a 3-axis reusability plan. Section 5 gives our conclusions and future work.

2. Requirements interaction management

Requirements interaction management (RIM) addresses the question how reused requirements might interact with each other in a common environment. It is very similar to feature interaction management used in the telecommunication domain in that they both try to detect possible interactions between features or components and provide guidance on how to resolve these interactions. For example, in the telecommunication domain, a very common feature is Call Forward on Busy Line (CFBL). With this feature, all calls to a subscriber's line are redirected to a predetermined number when the subscriber's line is busy [4]. Another common feature is Call Waiting (CW), which is "a feature that allows the subscriber to be notified of an incoming call while he is busy in a conversation and to accept the new call by putting the originating call on hold, then the subscriber is able to toggle between the two calls" [4]. Now if a user subscribes to both features and then tries to activate both of them at the same time, these two features will obviously negatively interact as they cannot be activated at the same time. Each of the two features requires a different action to be taken when someone calls this user and they are busy talking to someone else.

Unfortunately, research in feature interaction has focused mainly on telephony and has a very narrow view of requirements. In addition, the telecommunications domain is well defined and the requirements of it are reasonably well understood. Several problems caused by neglecting RIM when building a new system using reusable requirements have been reported in literature, ranging from minor issues to real disasters. The computers and risks literature [5], [6] has numerous examples of incidents caused by reusing existing systems in new situations, or by unanticipated interactions between systems.

3. A conceptual process framework for requirements reuse

Through our review of the current research on reuse at the requirements level, we were able to identify a conceptual process framework that was, more or less common to all approaches.

The development of this framework helped us in two ways: We gained a better understanding of how requirements can be reused. The second benefit, which we believe to be of great importance, was the identification of the different levels of abstraction for software requirements. For example, a requirement can be very specific to a certain system in a certain domain which thus gives it a very low level of abstraction and a low reuse potential; while another requirement can be very general, with a very high level of abstraction and thus a higher reuse potential. We identify the following abstraction levels: *domain-specific requirements*, *generic requirements, and domain-requirements frameworks*.

Domain-specific requirements are requirements that are derived from a certain domain for certain applications and are concerned only with this domain. Therefore, they are at a low level of abstraction as they cannot be reused in any other domains and applications (e.g. system-specific requirements).

General requirements are requirements that are with some variations common to different applications, i.e., one can replace the differentiation part in the requirement with a variable making it to a reusable requirement (e.g., The system shall support saving of email addresses up to X entries.)

A domain requirements framework is a framework that provides guidance on how to generate a reusable requirements specification document with hookups to facilitate development

The first part of our process for requirements reuse starts with the analysis of a group of systems (e.g. systems that are part of a product line) in a certain domain that have numerous commonalities between them, and thus it is promising to build a reusable platform for them. The second part of this process is the actual building of a new system that belongs to this specific domain. The developer will reuse requirements of the common platform to build the new system, along with introducing the new requirements which this specific system needs. The third part to this process is interaction management of the produced set of requirements. This general process framework can be seen in figure (1) and is described in more detail in the following subsections.

3.1 Phase 1: Elicitation

The aim of phase 1 is to create a database of reusable requirements that can be used later when developing a new system. Phase 1 starts with domain analysis during which a team of developers starts to analyze already developed systems that have numerous commonalities between them. There are many techniques that can be used to do the analysis such as commonality analysis [7] or design spaces [8]. The output of this analysis is a set of reusable requirements (kernel requirements and/or parameterized requirements and/or optional requirements) along with a set of rules of usage that the developer has to follow when reusing these requirements (e.g. requirements A and B are mutually exclusive). The next step is to gather each group of similar requirements into one component (e.g. all reusable requirements related to engine ignition are gathered into a component called engine ignition). Finally these components are stored in a database ready to be reused.

3.2 Phase 2: Reuse

The aim of this phase is to develop a complete set of requirements that describes the new system reusing some of the requirements stored in our database. This phase proceeds by tailoring the reusable requirements and adding also some new requirements that are needed to meet the specific needs of this new project. There are various approaches offered in literature that help the developers during this phase [9] [10].

3.3 Phase 3: Requirements Interaction Management (RIM)

We added this phase as we see it as a very important task that must be done to ensure that the reused and the new requirements will not have a negative influence on each other. The outcome of this phase is a set of requirements that have been analyzed for the detection of any interactions between them.

We suggest that this phase uses both offline and online approaches. An offline approach is for instance when the developer analyzes the requirements without building executable models of the system, whereas in an online approach models of the system are executed in a run time environment.
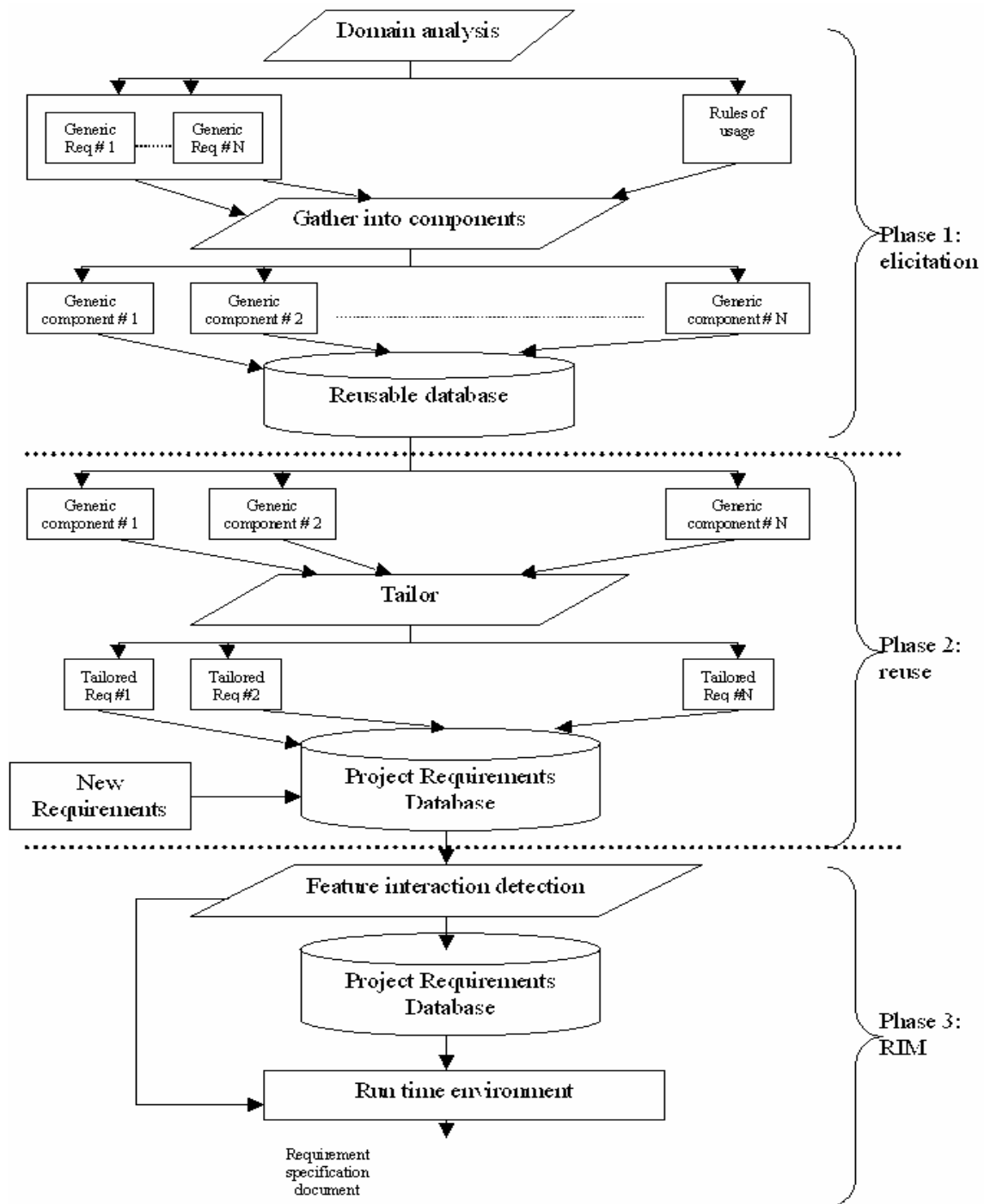

Figure (1): Requirements reuse framework

4. A new reusability plan

In section 2 we introduced the importance of requirement interaction management and in section 3 we introduced our understanding of requirements engineering reuse. In this section here, we combine requirements interaction management with the reuse of requirements through the introduction of a reusability plan which is an extended version of the typically used by software architects. Figure 2 shows the typical reusability plan on the left side, which focuses only on the design and implementation phases of the development of the product and which does not address interaction management. The right side of figure 2 presents our reusability plan in which we extended the original reuse plan to include the requirements engineering phase and added a third axis labeled "interaction management". The addition of a third axis required us to think what the different levels of abstractions of the different items are (e.g. frameworks, components, patterns, libraries, etc). For example, we can say that libraries, which belong to the implementation level, have a low level of abstraction and their interactions are difficult to resolve. This is because code libraries are normally very specific to certain applications and when different libraries from different places are glued together, most likely contradiction between them will be found (e.g. contradictions between input/output values, number of variables, global variables contradiction, etc). Even when the developer is able to find a conflict, its resolution is very difficult and requires the developer to look at the details of the code library.

As shown in figure (2), we concluded that as the level of abstraction goes higher, interaction management also becomes more difficult. For instance, detecting and resolving interactions between domain-specific requirements is easier than between generic requirements. Interactions are the most difficult to manage in domain-requirements frameworks. For instance, we can think that domain-specific requirements are very well known and defined; they were very well researched and studied already in previous systems, and the developers have complete details of these requirements. However, generic requirements are not fully specified and known.

For example, consider a distributed network environment in which there is a performance requirement stating that the system will be able to execute all requests for processing from different terminals in no more than 20 msec. Also consider another requirement stating that the system will be able to work with X units to produce an efficiency of Y %. Basically, there is no interaction between the two requirements in a normal situation. However, if X=10,000, there is obviously a clear interaction because the system will likely not be able to achieve any more the required response time of 2 msec. In the case of domain-requirements frameworks, in which the abstraction level is even higher and the developer can chose between different alternatives, RIM is even more challenging. It is possible that requirements may only interact after implementation decisions are made. How can this downstream effect be reflected in the original requirements?

This proposed reusability plan has various benefits. For example, this reusability plan provides a visual link between interaction management and the different levels of abstraction of the requirements. This visualization helped us in developing guidance on how to conduct interaction management during requirements reusability. It also introduces requirements engineering to be an area of reuse that framework developers must pay attention to.
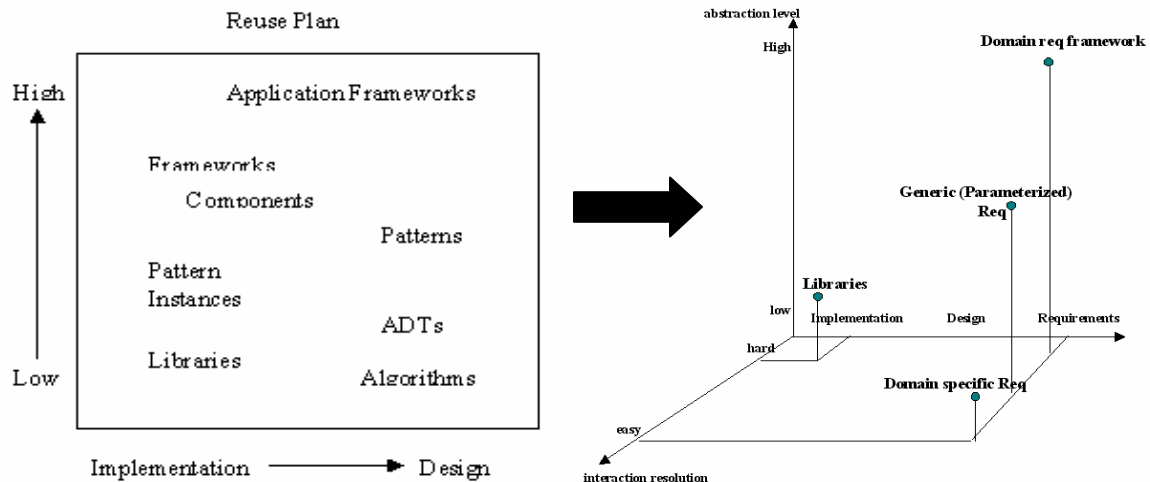

Figure (2): Requirement reusability plan

5. Summary and future work.

In this paper we presented our understanding of how requirements engineering reuse is carried out through a general framework that describes a common procedure that we were able to identify in various approaches. We combined both, requirements reuse and interaction management in a reuse plan that described what levels of interaction management the developer should expect at different levels of abstractions.

We are convinced that a requirements interaction management framework is very beneficial for anyone considering requirements reuse. For this purpose, our future work focuses on developing a comprehensive requirements interaction framework. This framework will be a three-level framework that helps detect different interactions at different levels of complexity. The first level of this framework can be used in situations where information on known interactions is already available in a knowledge base. The second level identifies likely (but not guaranteed) interactions between requirements based on requirements attributes. The third level requires the development of formal models of the domain, which is very time consuming but very efficient in the detection of feature interactions. This framework has the advantage that it can be used to detect known interactions with very little amount of effort as in the first level but it also can be used to detect full interactions using formal methods. This framework is outlined in [11].

References

[1] SPC (1992), Software Productivity Consortium Reuse, Adoption Guidebook, SPC-92051-CMC, November 1992.

[2] H. J. Hoover, P. G. Sorenson, G. Froehlich, A. G. Olekshy. (2000), Developing engineered product support applications. In Proceedings of the 1st Software Product Line Conference, sponsored by the Software Engineering Institute, Denver, CO, Aug. 2000, pp. 451--476. Published as Software Product Lines - Experience and Research Directions, P. Donahoe, ed., Kluwer Academic Publishers, 2000.

[3] W. N. Robinson. P. Pawlowski, S. Volkov, (1999), Requirements Interaction Management, Georgia State University, Atlanta, GA, August 30, 1999.

[4] M. Calder, E. Magill, (2000), Feature interactions in telecommunications and software systems VI , IOS Press, ISBN 1586030655.

[5] N.G. Leveson, (1995), Software: System safety and Computers, Addison-Wesley Pub. Co. Inc.

[6] P. Ladkin, (1995), In The Risks Digest, P. G. Neumann (Ed.), ACM, 15.30, December, 1995.

[7] M. Ardis, D. Weiss, (1997), Defining Families: The Commonality Analysis, in Proceedings of the Nineteenth International Conference on Software Engineering, pp. 649-650, May 1997.

[8] L. Baum, M. Becker, L. Geyer, G. Molter, (2000), Mapping requirements to reusable components using design spaces, In Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00), 19-23 June 2000, Schaumburg, Illinois, USA.

[9] W. Lam, (1997), Achieving Requirements Reuse: A domain-Specific Approach from Avionics, Journal of Systems Software, 1997: 38: 197-209.

[10] M. Mannion, B. Keepence, H. Kaindl, J. Wheadon, (1999), Reusing Single Requirements From Application Family Requirements, 21st International Conference on Software Engineering (ICSE'99) Los Angeles, CA, May 1999. pp 453-462 ISBN 1-58113-074-0.

[11] M. Shehata, A. Eberlein, (2002), Requirements Interaction Management: A Multi-Level Framework, To appear in SEA 2002 - The 6th IASTED International Conference on Software Engineering and Applications. MIT, Cambridge, USA - November 4-6, 2002.