Using Acceptance Tests For Incremental Elicitation of Variability in Requirements: An Observational Study

Yaser Ghanam and Frank Maurer
Department of Computer Science
University of Calgary
Calgary, AB, Canada
{yghanam, fmaurer}@ucalgary.ca

Abstract—Variability in software systems refers to the notion that the components constituting the software may vary due to a range of factors including diverse customer needs, technical constraints, and business strategies. Traditionally, variability has been treated proactively by investing in an upfront domain analysis phase. Such proactive treatment of requirements is not encouraged in agile environments. This paper provides an observational study examining a reactive approach to variability wherein acceptance tests are used to elicit variability from requirements in an incremental manner. The findings suggest the following: the approach does support the evolutionary nature of agile development; the approach is easy and quick to learn; using acceptance tests yields consistent variability interpretations; and acceptance tests – on their own – may be insufficient to reflect implicit variability constraints.

Keywords – acceptance tests; variability; feature models; software product lines; reuse

I. INTRODUCTION

During the past decade, iterative models, including Agile Software Development (ASD), have proved to be more realistic and effective than traditional models in achieving fast delivery and incorporating customer’s feedback in every cycle of development [1]. Another remarkable advancement in software engineering has been software reuse [9]. Instead of building software products from scratch, assets that were produced in previous software projects could be enhanced and reused. This offered a great potential in terms of improving productivity and quality [12], especially when reuse is achieved at the product level as evident in Software Product Lines (SPLs). A SPL is a family of software-intensive systems that share a common set of features while allowing for a margin of variability to satisfy different customer needs [3]. SPLs deal with similar systems as a family of products sharing a library of core assets. But since customer requirements are rarely exactly the same, shared assets have to accommodate a certain degree of variability.

Having realized the advantages of ASD as an iterative paradigm and SPL engineering as a reuse paradigm, it is curious to know whether the two can be combined to achieve reuse and variability while maintaining the main themes of agility such as iterative development and minimum upfront design. In our other work on the subject [14], we proposed an approach that leverages acceptance tests (ATs) to elicit variability in requirements in an evolutionary manner (a summary of the approach is included in this paper). We evaluated the feasibility of this approach using a case study. In this paper, we describe an observational study that we conducted to explore the strengths and weaknesses of the approach using the help of independent participants.

In the following section, we provide some background on the definition of variability. Section III provides a brief survey of related work in the literature. Section IV presents a summary of the incremental approach. In Section V, we present the observational study. We discuss the results of the study in Section VI. Finally, we conclude in Section VII.

II. BACKGROUND

Variability in software systems refers to the notion that the components constituting the software system may vary due to a range of factors including diverse customer needs, technical constraints, and business strategies. According to the Orthogonal Variability Model (OVM) by Pohl et al. [13], variability in a product line is described by a number of variation points, a set of variants for each variation point, and possibly some constraints. A variation point is an aspect in a certain requirement that can have multiple states of existence in the system. Each state of existence is called a variant. The selection criteria of variants might be governed by some constraints. For example, let’s say that a module called Weather Watch consists of three aspects, namely: Weather Model, Weather Trend Analyzer, and Weather UI Panel. Both the Weather Model and the Weather UI Panel represent mandatory aspects without which the Weather Watch model would not be useful. On the other hand, considering a number of factors such as cost and computation capability, the Weather Watch may or may not have the Weather Trend Analyzer aspect which makes this aspect optional. Optionality can be expressed as a variation point for which two variants are defined:

Variation point VP1 – Existence of the Weather Trend Analyzer:
Variant V1 – Trend Analyzer exists.
Variant V2 – Trend Analyzer does not exist.
Governed by the constraint:

Constraint C – V1 and V2 are mutually exclusive.

Besides optionality, when multiple alternatives could be selected for a given feature, this variability can also be
expressed as a variation point for which there might be two or more variants. In this example, depending on the type of hardware platform running the Weather Watch module, two variants are defined – only one of which can be selected for a given system:

**Variation point VP2** – Panel type:
- **Variant V1** – Handheld panel.
- **Variant V2** – PC panel.

Governed by the constraint:

**Constraint C** – V1 and V2 are mutually exclusive.

If more than one variant can be selected for a given feature, the constraint is defined using a minimum-maximum format. For example, in the Security Manager feature, the customer can select one, two or all three of the supported access control technologies – namely, V1: PIN-protected locks, V2: access by magnet cards, and V3: fingerprint authentication. Therefore, the constraint would be:

**Constraint C** – [1..3] multiplicity imposed on V1, V2, V3

Variability, as defined above, can be modeled using feature modeling techniques which have become an essential aspect of SPL engineering. A feature model is a representation of the requirements in a given system abstracted at the feature level [15]. In SPLs, feature models represent a hierarchy of features and sub-features in a product line and include information about variability in the product line. In our work, we use a common modeling technique called Feature-Oriented Domain Analysis (FODA) [10]. FODA models features in a given system in what is called a feature tree and uses a specific notation to describe variability as explained in the following example. Consider the simple feature tree in Figure 1a that represents the Weather Watch module. Both the Weather Model and the Weather UI Panel are mandatory features – which is denoted with a solid line. On the other hand, as discussed previously, the Weather Trend Analyzer is optional, which is denoted with a dashed line. The Weather UI Panel can have one of two different formats depending on whether the application is to run on a handheld device or a normal PC. This is denoted in the tree as an arch. By looking at this tree, one can deduce the different feature instances that can be produced from this generic module such as the ones shown in Figure 1b.

![Figure 1 - Using a feature tree to model variability](image)

### III. RELATED WORK

Variability management is a key concept in SPL engineering. There is a fairly large body of research investigating efforts to improve analysis, modeling and realization techniques of variability. For instance, Coplien et al. [4] proposed a model called FAST to identify, analyze and document scope, commonality and variability. In their model, information is to be elicited and documented about the domain, predicted commonalities, and parameters of allowed variation. They also discuss concepts such as procedures, inheritance and class templates to realize variability in the code. They do not, however, consider the evolution of members in the family. A later effort by Gurp et al. [6] suggested a feature model to manage variability, where a feature is a set of logically grouped functional and nonfunctional requirements. This is somewhat similar to the basis of our work, but we use ATs to represent features as opposed to traditional textual representation of requirements. Buhne et al. [2] proposed an alternative model (later called OVM) that was based on an explicit representation of variability through variation points and variants. To communicate variability to customers, Halmans et al. [7] proposed extensions to use-case diagrams. Our work aims to achieve a similar objective, but the use of ATs is not limited to communicating variability with the customer, but is also a means of managing this variability. The use of change-sets in SPLs was originally proposed by Hendrickson et al. [8] to model product line architectures. The model, nonetheless, was not meant to help in eliciting or communicating variability. Rather, it was intended for architecture modeling.

### IV. SUMMARY OF THE INCREMENTAL APPROACH

In this section, we provide a summary of the proposed incremental approach to elicit variability in requirements using ATs. The full details of the approach and its feasibility evaluation are available in [14]. The approach relies on the AT layer as a medium of communicating requirements between the customers and the developers of a particular system. ATs proved to perform well in communicating requirements amongst stakeholders [11]. A test page typically envelops a number of AT tables that test a particular feature. The approach can be summarized in six points, namely:

1. The very first system is built in a normal ASD process to satisfy the requirements of the customer at hand without investing into future speculations of what may vary.
2. An initial feature model of the system is produced using ATs as the building units.
3. Upon the demand of a similar system by a new customer, the existing feature model and the associated ATs are made available to the customer.
4. The new customer picks those ATs that meet their specific needs.
5. If the current feature model cannot satisfy the customer’s needs, the customer defines a change set to add, remove or replace ATs.
6. Based on the change set produced in 5, the feature model is updated.

To illustrate the previous points, we use the example of an intelligent home system. In an intelligent home system, test tables in a page describing an access control feature looks like the one in Figure 2.
This test page looks almost the same as a traditional FIT test page. The only difference is that we denoted some test as “default” and others as “optional.” Default artifacts are those that are essential to reflect the value of the feature at hand. Optional test artifacts, on the other hand, are those that can be looked at as add-ons rather than necessities. This might be perceived differently by different customers. A feature model is built for the current feature consisting of the two tables A and B as shown in Figure 3.

Upon the demand of a similar system by a new customer, the customer is presented with the feature model that reflects the test page in Figure 2. Now, say that when the new customer was presented with the feature model, they requested a change to the access control feature via PIN. The customer then has the option to exclude existing tables or add new ones. The customer requests the needed customizations using a change set to add, remove or replace tables as shown in Figure 4.

Table C is added to the test page as one more option future customers can pick from. However, the addition of Table D is not as straightforward due to its conflict with Table B. That is, according to Table B, the input should be locked for 2 minutes after 2 failed attempts. Whereas according to Table D, the user is allowed 3 attempts after which the owner is notified. To solve this issue, we can impose a constraint that Table B and Table D cannot coexist. The new version of the test page yields the feature model shown in Figure 5. In this case, a [0..1] constraint indicates that only one table may be selected amongst the set {Table B, Table D}.

### V. OBSERVATIONAL STUDY

#### A. Goal

The goal of the study presented here is to examine the foundations of the approach with the help of independent participants according to criteria deduced from the merits of ASD. Namely, we were interested in examining four aspects: evolution, learnability, consistency, and constraints. To examine these aspects, we conducted an observational study involving 16 graduate students. All the participants had some background in software engineering. Participants were asked to study a written tutorial on the approach, solve three exercises, and then fill out a questionnaire. Table 1 lists the aspects and the required observations to study these aspects.

<table>
<thead>
<tr>
<th>Q</th>
<th>Aspect</th>
<th>Required Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Evolution</td>
<td>We will observe if the participants will be able to start at an initial state of the feature and incorporate new requirements as they come. The final state of the feature should be consistent with the intended one.</td>
</tr>
<tr>
<td>2</td>
<td>Learnability</td>
<td>After going through a written tutorial, we will observe if the participants will be able to: a. Distinguish between a feature instance and a generic feature model. b. Use ATs as building units for feature models. c. Relate instantiation requests to the required ATs.</td>
</tr>
<tr>
<td>3</td>
<td>Consistency</td>
<td>We will observe if the participants will be able to build hierarchical models that are consistent with the anticipated ones (hence, consistent across different participants).</td>
</tr>
<tr>
<td>4</td>
<td>Constraints</td>
<td>We will observe if the participants will be able to deduce all explicit and implicit constraints from the provided ATs.</td>
</tr>
</tbody>
</table>

### VI. RESULTS & DISCUSSION

#### A. Tutorial and Exercises

The performance of participants was measured by the time spent on each exercise, and the consistency of their outcome compared to the outcome we had anticipated. Exercise 2 took almost double the time of Exercise 1. This is normal considering that Exercise 2 was a more complex one. Nonetheless, the consistency did not change much. We attribute this to the learning effect that is likely to have occurred during Exercise 1. Except for one participant who could not understand Exercise 3, all participants could solve Exercise 3 with 100% consistency.

Figure 6 provides a closer look into the performance of all 16 participants in exercises 1 and 2. Each dot represents one participant. The y-axis represents the percentage of...
consistency (all data points are above 60%). In both cases, we observed that participants were in fact able to start at an initial state of the feature and elicit variability from new requirements as they came. The final state of the feature was consistent with the intended one in more than 80% of the cases. Since Exercise 1 was the first exercise to follow the tutorial, we chose to analyze it in more detail. We found that all participants could achieve the three objectives mentioned under the learnability aspect. To check for the consistency aspect, we combined the results of all three exercises and we found that the interpretations of the contents of the ATs were mostly consistent amongst more than 80% of the participants. We did, however, find some discrepancies in the way participants chose to model certain parts.

Figure 6 - Consistency (%) in Exercise 1 to the left, and consistency (%) in Exercise 2 to the right

When looking at realizing and modeling constraints, we found little evidence that ATs were sufficient to deduce implicit constraints. Although all but one participant could realize and model explicitly mentioned constrains, half of the participants could not deduce an implicit constraint we had planted in Exercise 2. There were also some instances where constrains were unnecessarily imposed.

B. Questionnaire

According to the questionnaire, participants found the approach flexible and easy to grasp and apply. Figure 7 shows the responses to the Likert-scale questions. Dealing with ATs seemed to be a bit of a hassle, but only for participants who had not worked with ATs before. One participant, who did work with ATs for a while, noted that he “found dealing with ATs very easy.” Other comments mainly reflected the trickiness of dealing with constraints. Some participants also commented on the scalability of the approach for larger and more complex systems.

Figure 7 - Responses to the questionnaire

VII. CONCLUSIONS

SPL engineering is an increasingly important paradigm in software development. Nevertheless, using SPLs with ASD practices require a special adaptation scheme. We proposed an approach to provide such a scheme by utilizing test artifacts to elicit variability in requirements. The observational study presented in this paper examined four aspects of the proposed approach and provided interesting insights into its strengths and weaknesses.

REFERENCES