Extreme Product Line Engineering: Managing Variability & Traceability via Executable Specifications

Yaser Ghanam
University of Calgary
yghanam@ucalgary.ca

Frank Maurer
University of Calgary
frank.maurer@ucalgary.ca

Abstract

Extreme Programming (XP) has been reported to work well by valuing principles of simplicity, lightweight practices, effective feedback and continuous process and product improvement. This paper describes an approach towards managing software product lines in a setting where XP practices are common. The paper is an action research describing a case where we handled variability in the domain of intelligent home systems to satisfy a range of requirements by our industrial partner. The paper delves into how variability and traceability of requirements can be managed via executable specifications. A case study was used to evaluate the approach, and it provided initial insights on its feasibility and usefulness.

1. Introduction

1.1. The Problem Context

Intelligent home systems make it possible to monitor and control the surrounding environment in a smart home. These systems usually need to encompass a spacious variety of home infrastructures, devices, security mechanisms and many other aspects. Every home has its own floor plan and hardware capabilities; and on top of that, every home owner has different interests in what needs to be monitored or/and controlled in the home. Furthermore, even the interface - through which the end users control and monitor the home - is likely to have a raft of possible designs and technologies. We stumbled upon this issue of variability as our XP team (in the software engineering lab) was developing a smart home solution for an industrial partner. The problem was that many variation points existed, each of which had to embrace a handful of variants. We realized that developing a separate system for each home was a viable option but not an economically appealing one. The challenge was to be able to deliver a system that only had those variants requested by a specific customer without substantial rework. Given that these systems had a lot of overlapping requirements, the apparent solution to our problem was to adopt a software product line practice. Achieving this without affecting the agility of our team was the dilemma we tried to solve.

1.2. Software Product Lines

A software product line (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 [1]. Companies consistently report that SPLs yield significant improvements in productivity, time to market, product quality, and customer satisfaction [2]. Commonality between systems is what makes SPLs economically effective; whereas variability is what makes mass customization possible. 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. For instance, the customer of an intelligent home system should be able to choose a subset of components that fulfills his wants. Furthermore, it should be possible for customers to tailor certain aspects of these components to meet their specific needs. A security system, for example, offers different techniques to secure access control such as PIN protected locks, access by magnet cards and finger print authentication. When choosing to have a security system component, customers may select one or more of these options. Traditionally for SPLs, commonality and variability are documented in requirement artifacts as well as multilevel design artifacts. These artifacts trace all the way down to code units (i.e. packages, classes, methods) so that reuse can be achieved and customization is done in the right place. There are three main factors at play in SPL engineering: 1) Commonality and variability management: eliciting and communicating commonality and variability in

requirements to stakeholders; 2) Traceability of commonality and variability from the requirements to the code; and 3) Managing and tracking reuse of code across different instances of the system, usually driven by the previous two elements.

Our research is focused on investigating how agile organizations, especially those adopting XP, can benefit from SPL engineering. We developed a model based on test-driven development (TDD) that utilizes executable specifications[1] (ES) to achieve the three abovementioned elements. This paper specifically delves into the first two elements.

1.3. Preliminary Analysis

Normally, SPL engineering starts off with a phase called domain engineering. During this phase, engineers plan for products as a family rather than as individual instances. Domain engineers conduct commonality and variability analysis to produce a variability model. This analysis is conducted through a variety of techniques. In this paper, we discuss the technique by Pohl et al [3] - it entails four major steps:

1. Define common requirements: use application requirement matrices, priority analysis or checklist based analysis to review the requirements of systems you have previously built or you expect to build in the future. Extract repeated requirements, requirements to become common in the future, or strategically common requirements.
2. Define requirement variability: look at how requirements across different systems might vary and understand why they vary. The objective of this step is to extract variation points, possible variants, as well as any dependencies or constraints.
3. Document findings in (1) and (2): this produces domain requirement documents that explain to application engineers how to instantiate applications.
4. Proceed to the next phases: use the documentation produced in (3) to design, implement, and test the architecture and its constituents.

This approach presumes sufficient knowledge about the domain and the needs of the market. It also requires a substantial amount of work for upfront analysis – which goes against core principles and believes of XP. The approach has proven to work well for organizations under certain assumptions. Table 1 lists these assumptions and shows how they were in conflict with the practices of our XP team.

The following section provides a thorough discussion of our approach. In Section 3, we present a case study and discussion of the model. Section 4 is a review of related work. We draw our conclusions in Section 5.

2. Extreme Software Product Lines

2.1. Organizing Test Artifacts

The previous section showed how variability analysis is conducted in some traditional SPL practices, and how a number of the basic suppositions underlying these practices are not suitable for an XP culture. In this section, we present a model bridging the gap between SPLs and XP. This model addresses the notion that XP does not produce elaborate requirement documents to describe the system under development. XP, however, produces other artifacts (i.e. ES) that can alternatively be used to describe the system and act as anchor points for traceability relations. In story TDD, ES are written before writing code in the form of acceptance tests (AT). ATs are usually written collaboratively by the stakeholders to ensure a consistent understanding of the system. These tests can be automated by tools like FIT[4].

Table 1. Assumptions of the traditional model of variability, and conflicts with XP practices

	Assumption	Conflict
A	The organization has built systems in the same domain. Or sufficient knowledge about the domain is elicited upfront. Pohl et al [3] assert that building an SPL "requires sophisticated domain experience."	This implies that adopting a product line approach might be infeasible for small organizations entering a new market. Moreover, XP considers acting upon predicted future requirements too risky, and thus may not be willing to substantially invest in domain requirement elicitation upfront.
B	Traditional requirement engineering was done for each system in **"A"**: elicitation, negotiation, validation, documentation, management.	In XP, development starts immediately. As for requirements, XP does not dedicate a requirement engineering phase, but rather preaches a minimalistic way of obtaining customers' needs using story cards.
C	Requirement documents resulting from **"B"** are available and up-to-date. They accurately map to and are consistent with design, code and test artifacts.	In XP, unless requested by the customer, requirement, design and test documents are considered of less value than actual implementation. In case documentation exists, it is generally difficult to ensure documents are up-to-date and consistent. Most XP teams will not create requirement and design documents to the extent expected in SPL engineering.

[1] ES is a general term that refers to what is known as story tests or executable acceptance tests.

This makes it possible to continuously run these tests against the code developers write to measure how complete a feature implementation is. We specifically propose the use of ES in the form of ATs to model variability in product families. The benefit of using ATs is twofold. For one, no burden is added on the XP team to produce ATs given that ATs are a natural starting point in XP iterations. Secondly, since XP promotes a refactor-whenever-needed notion, these tests are continuously updated to reflect changes in the system. Hence, we can assume these artifacts represent a sufficiently up-to-date account of the system they test.

In order to use test artifacts as a basis for our model, it is important to understand in what form these artifacts exist in our repository. We considered the use of a common tool to write and run ATs called FitNesse [5]. FitNesse is an AT framework based on a fully integrated standalone wiki. With the help of the user guide provided with the FitNesse tool package, we procured an object model that reflects how test artifacts relate to the system under test (SUT) and to each other. As Figure 1 shows, the production of test artifacts is driven by features requested by the customer. In this paper, we use the term feature to refer to a chunk of functionality that delivers business value [6]. There is no restriction on how small or large this functionality is, as long as the customer thinks its existence would add value to the delivered system. Internally, nonetheless, developers may choose to break the feature down into sub-features to make it more manageable and testable. While one or more test artifacts are produced to test a single feature, it is also true that a single test artifact might cut across a number of features in the system.

A test artifact can exist at different granularities. Typically, developers would start by creating a test project for the SUT. The test project has a number of test suites that are optionally used to organize tests into a recursive folder-like structure. Grouping tests into suites might be based on a feature breakdown or might be chronological based on XP iterations. Each suite consists of one or more test pages. In FitNesse, these pages are files, each of which has a number of tables representing user stories. Test tables can take different formats based on the type of fixture they are linked to (e.g. column or row). In essence, these tables are the specifications of the customer. In order for test tables to be executed, they are linked to a thin layer of testing code called a fixture. It is within these fixtures where the actual production code is tested. A fixture uses a number of code units to execute specifications from the AT tables. According to this model, capturing commonality and variability in features can occur at different granularities of test artifacts. Some test artifacts can be seen as common across different applications in the family, and thus are considered default artifacts. Some other artifacts may be described as optional or alternatives. For example, a customer might want to exclude a certain scenario or include an additional one in a given feature. In this case, variability is defined at the test page level to include, exclude or add certain test tables. Some of these tables may be in conflict; therefore, multiplicity and dependency constraints need to govern the selection process. The following subsection will illustrate this concept further.

2.2. Introducing Variability

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 FitNesse 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. If removed, the feature becomes meaningless or valueless. Some other tables like setup tables might also be considered default if their existence is a prerequisite for other default tables to execute. The default attribute should not constrain the flexibility of responding to new requirements. It is only an indication, for new customers, that this element was of special importance to previous customers, making it a good candidate to become common across different instances. 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. Therefore, optionality is only a guide for future customers that an element might be cut out without omitting the value of the feature. This initial assumption might be challenged later on by other customers who deem the optional element to be an indispensible part of the feature. Thus, an optional test artifact could be upgraded to become a default one and vice versa. Now, say a new customer requests a change to the access control feature via PIN. The customer is given the test page in Figure 2.

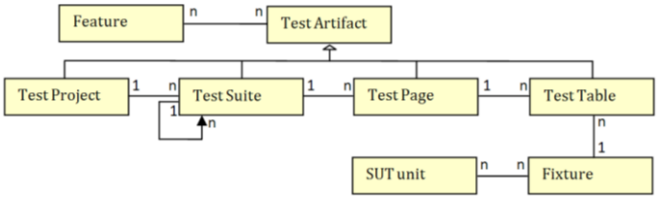

Figure1. An object model for test artifacts

Default	Setup		
Add resident	John Smith	PIN	12345

Default	Table A. Authentication through keypad input			
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	True		

Optional	Table B. Input locked after two failed attempts			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	22222
Check	Door is unlocked	False		
Check	Input locked	True		

Figure 2. A test page is composed of a number of test tables

He has the option to exclude existing tables or add new ones. The customer requests the customization shown in Figure 3. 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. We can visualize the new version of the test page using a commonly used concept in SPLs called a feature model [7] as shown in Figure 4. A solid line symbolizes a default artifact whereas a dotted line symbolizes an optional one. Multiplicity constraints in the form of [min..max] may be added to govern the selection of artifacts. In this case, a [0..1] indicates that only one element may be selected amongst the set {Table B, Table D}.

- *Replace Table B with Table D (cannot coexist due to conflicting behavior).*
- *Add Table C.*

Optional	Table D. Owner notified after three failed attempts			
Enter	Resident name	John Smith	PIN	11111

Optional	Table C. A successful attempt after a failed one should prompt the user to enter his info again			
Enter	Resident name	John Smith	PIN	11111

Figure 3. Customization requested by the customer

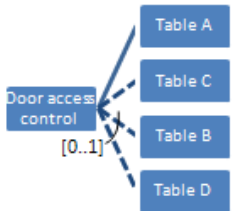

Figure 4. A feature graph representing variability

2.3. Instantiation Process

The instantiation step relies on the notion that ATs ideally serve as an accurate and up-to-date reference of features in the core system. A core system is one that continuously accumulates assets produced towards the satisfaction of previous customer requests (as explained in the previous subsection). It is from the core system that family members are produced as variants in the product line. The discussion to follow assumes that a core system is available and is represented through a library of ATs organized as discussed in the previous subsection. The instantiation process requires a number of steps as shown in Figure 5, namely:

1. **Select ATs:** upon a new request of the system, the customer is provided with ATs that embody the different capabilities (features) currently available in the core system. Customers are to select only those ATs that match the criteria (scenarios) they are looking for (highlighted in Figure 5). The outcome of this step is a subset of ATs.
2. **Execute ATs:** the selected subset of ATs is run against the core system; and a test coverage report is obtained using a test coverage tool. The coverage report provides information about what code units or fragments were used to execute the given subset of ATs. This includes modules, namespaces, classes, methods, and files in both the testing code (fixture code) and the tested code (production code).

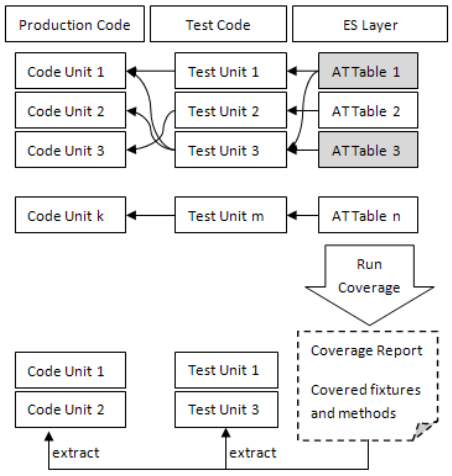

Figure 5. Variant instantiation

3. **Extract code:** based on the coverage information provided in step 2, relevant code units and fragments will be extracted from the core system. Any fragments that are not needed in the current

instance are eliminated. This is the most complex and crucial step as will be discussed later. The outcomes of this step are two, namely: a new system that represents a variant of the core system, and a new test suite that possesses the fixture code needed to provide test coverage for the new system.

4. **Verify and build:** in this step, the newly instantiated system is compiled and built to make sure the extraction step did not produce any flaws in the code or the references. Then, by utilizing the test suite extracted in the previous step, the selected subset of ATs (from step 1) is run against the new system to verify the satisfaction of acceptance criteria within the new variant.

2.4. Change Management

New needs: In the previous section, it is implicitly assumed that customers' needs can always be fulfilled by existing features in the core system. Alas, in reality this is not the case. Customers usually introduce new needs, especially in agile contexts where customer involvement is key, and where an incremental approach is encouraged. For any new requests in the system, an AT needs to be written to represent the request at hand, and then is added to the library of ATs in the core system as per the variability model described earlier. In case the customer requires a change in a detail of a previously written AT, modifications better happen on a new copy of that AT (rather than the original AT). The new copy of the AT is to be treated as an entirely new AT representing a new option for customers to pick from. In all cases, developers implement the required functionality until the AT passes – basically, development takes its normal course as it would in a typical agile project. In case a conflict is introduced by the change (i.e. a new scenario cannot coexist with an existing one), it should be resolved in a way that does not affect other variants in the family. If this is not possible, the conflicting change could be explicated in the variability model by imposing constraints to govern the instantiation process. Managing conflicts in the variability model is beyond the scope of this paper.

Maintenance: during the development cycle, some modules endure code improvements, refactoring, restructuring and bug fixing. These changes can be categorized as external or internal changes. An external change is one that is visible to the customer (e.g. change in the sequence of operations). Conducting changes in this category requires changes in the corresponding ATs. And since test coverage information provides traceability to the code units affected by the change, this information is used to indicate where in the core system (not the variant) a change needs to occur. On the other hand, an internal change is one that affects the code but does not directly influence the customer (e.g. using web services instead of http requests). This kind of changes does not require an update of ATs. In both cases, after the change has been incorporated in the code, all variants in the family need to be re-instantiated and tested to make sure the change has propagated to all relevant variants and did not have a destructive effect on other variants.

3. Evaluation: Case Study

In the previous section, we presented an approach to manage variability and instantiate products using ATs. Here, we present an evaluation aimed at investigating the feasibility and usefulness of the approach. We use Action Research (AR) [8] to self-evaluate our approach against the original problem. AR is a well established evaluation technique in applied research. We applied the approach on the intelligent home system mentioned earlier (aka eHome). eHome is an application to monitor and control smart homes. So far, eHome has around 100 classes divided into model, view, controller, hardware, and communication layers. There are about 70 test cases covering about 90% of the model code. We encountered a number of variation points such as: interface touch capabilities (e.g. single touch versus multiple touch), interface orientation (e.g. vertical vs. horizontal), required modules (e.g. light control modules, RFID item tracking). In this section, we illustrate our idea by presenting an example treatment of a variation point from our case study.

3.1. Core System Status

Our customer requested a feature that would enable the end user to define macros to control devices at the home. A macro is a sequence of actions to be executed on demand. The feature was defined as per the AT shown in Figure 6.

setup two devices			
create macro	Relax Mode		
Check	*number of actions*	0	
add action	1	*level*	90
Check	*number of actions*	1	
add action	2	*level*	20
Check	*number of actions*	2	
Check	*device status*	1	0
Check	*device status*	2	0
execute actions			
Check	*device status*	1	90
Check	*device status*	2	20

Figure 6. AT for adding unconditional macros

A later request from the customer was to extend the previous feature so that it is possible to optionally constrain the execution of some macros by a set of conditions. These conditions are to be defined by the end user. Figure 7 shows the AT for this request. This AT was added to the same test page as the previous AT because there was a lot of overlapping functionality. A thin layer of fixture code was developed to execute both tables. Figure 8 shows what the contents of this layer look like. Production code units that made both test cases pass is shown in Table 2[2]. To summarize, the "macro addition" feature existed in the core system in such a way that the two ATs were supported. Both the fixture code and the production code incorporated the requirements of both scenarios.

3.2. Instantiation

In the previous section, the fact that a customer requested to add a certain extension to the feature (as per the second AT) does not mean that the feature has to exist in its fullest version in all variants of the system. Some customers would not like the complication of dealing with rules to constrain macros, and thus are satisfied with the simpler version represented by the first AT only. In our approach, customers communicate their preferences through the selection of ATs - Customers can choose the scenarios they would like to see in the feature, and may exclude some other scenarios that are unneeded. Therefore, for this feature we define a variation point that minimally yields two variants: one that supports the addition of unconditional macros only (simple version), and another that supports the addition of both unconditional and conditional macros (complex version).

The dilemma is to produce just-enough code to instantiate each variant. According to the proposed approach, the selection of ATs is the first step towards this objective. We distinguish two cases: in Case I, the customer chooses the AT in Figure 6 only. In Case II, the customer chooses both ATs. Table 3 shows the coverage results of executing the tests in both cases. We only show method coverage to simplify the analysis. A '✓' symbol besides a method means the method was visited when the AT was executed. 'P' stands for production code and 'T' stands for test code. Having access to the coverage information, the next step in the instantiation process is extracting the needed code units for the specific variant of interest.

To illustrate, consider Case1 (aka variant 1) where some methods are unneeded for a successful execution

[2] Primitive getters and setters, constructors and other auto-generated units are removed due to space limitation.

setup two devices					
create macro	Auto light				
add property	number_of_people	*value*	2		
Check	number of properties	1			
Check	*number of conditions*	0			
add action	1	*level*	90		
Check	*number of actions*	1			
add action	2	*level*	20		
Check	*number of actions*	2			
Check	*device status*	1	0		
Check	*device status*	2	0		
add condition	number_of_people	*operation*	BETWEEN	*operandA*	1
Check	*number of conditions*	1			
execute conditional actions					
Check	*device status*	1	90		
Check	*device status*	2	20		
setup two devices					
Check	*device status*	1	0		
Check	*device status*	2	0		
change property	number_of_people	*value*	0		
execute conditional actions					
Check	*device status*	1	0		
Check	*device status*	2	0		

Figure 7. AT for adding conditional macros

```
namespace TestingPro
{
    public partial class ActionsTestFixture : DoFixture
    {
        DeviceList deviceList;
        Rule rule;
        PropertyList propList;
        public ActionsTestFixture(){...}
        public void setupTwoDevices(){...}
        public void createMacro(string description){...}
        public bool addActionLevel(string id, int level){...}
        public int deviceStatus(int id){...}
        public void executeActions(){...}
        public void executeConditionalActions(){...}
        public void addPropertyValue(string attribute, int value){...}
        public void changePropertyValue(string attribute, int value){...}
        public int numberOfActions(){...}
        public int numberOfProperties(){...}
        public int numberOfConditions(){...}
        public void addConditionOperationOperandAOperandB(string property,
                                Operand operation, int opA, int opB){...}
    }
```

Figure 8. Fixture code for the core feature

Table 2. Production code units for the core feature

Class	Methods
Property	No class-specific methods
Condition	isTrue() : Boolean
Rule	getConsequenceAt(Int32 index) : Property addCondition(Condition condition) : Boolean addConsequence(String deviceID,Int32 level) : Boolean isSatisfiedBy(PropertyList properties) : Boolean getConditionCount() : Int32 getConsequenceCount() : Int32
PropertyList	getProperty(String attribute) : Property addProperty(Property property) : Boolean hasAttribute(String attribute) : Int32 getCount() : Int32
DeviceList	getDeviceWhereID(Int32 id) : Device

of the selected AT. In this case, the production code units needed are shown in Table 4; and the generated fixture code will be as in Figure 9. As shown in the table, in some cases, all the methods in a given class are not needed. This might imply that the class itself is to be abandoned. Nevertheless, as will be discussed later, this is not always a trivial decision.

3.3. Discussion

3.3.1. Insights from the evaluation. The objective of our evaluation was to check the feasibility and usefulness of our approach. It was a sanity check to make sure the approach can actually be employed in real settings. For this reason, the approach was applied on a real system. During the system development, we treated a number of variation points, but we only had the space to present a simple example. The initial insights from our evaluation indicate that the approach is indeed feasible. It is simple, lightweight and based on test artifacts which are naturally produced in XP contexts. The fact that it supports incremental extensions and change management makes it a good fit for XP practices when compared to heavyweight, big-design-upfront-based approaches. The results of the evaluation also underscore the usefulness of the proposed approach. That is, instantiating different products from a library of core assets - based on specific customer needs - promises to reduce cost and drastically improve quality and time-to-market. This is especially significant when no huge investments are required upfront, and a faster ROI is expected [9] – which are advantages this approach offers. Furthermore, the approach operates on the ES layer which makes the instantiation process agnostic to the testing or implementation language. We strongly believe the approach will provide great benefits for software practitioners, but its generalizability is constrained with the limitations discussed below.

3.3.2. Limitations. It is imperative to point out a number of issues we encountered during our experience. We think the accuracy of the code extraction step can be improved – especially when treating fairly complex code – by taking into consideration coverage reports at different granularities (e.g. classes, methods, namespaces) and in different forms (e.g. branch coverage, visit coverage, sequence coverage). In our evaluation, we only discussed visit coverage for methods. We could see in Table 3 that even though some methods and classes were not needed anymore, that did not necessarily imply simple removal of these units and their references from the code assembly. In our case simple removal did the job. But in some other cases, the method might be shown to be uncovered because it is referenced in a condition block that was not executed. This tends to indicate that the uncovered branch is not needed anymore because there is no match for the case in the test artifacts. But to ensure a fair treatment of such cases, use of branch coverage or even static code analyzers might be required before taking a decision. This issue will be further researched in the near future. Moreover, so far we only evaluated our approach on model-based

Table 3. AT coverage report

ID	Method Signature (as given by the coverage tool)	Type	Case I	Case II
1	isTrue() : Boolean (in Condition)	P		✓
2	getConsequenceAt(Int32 index) : Property (in Rule)	P	✓	✓
3	getProperty(String attribute) : Property (in PropertyList)	P		✓
4	addCondition(Condition condition) : Boolean (in Rule)	P		✓
5	addConsequence(String deviceID,Int32 level) : Boolean (in Rule)	P	✓	✓
6	addProperty(Property property) : Boolean (in PropertyList)	P		✓
7	getDeviceWhereID(Int32 id):Device (in DeviceList)	P	✓	✓
8	isSatisfiedBy(PropertyList properties) : Boolean (in Rule)	P		✓
9	hasAttribute(String attribute) : Int32 (in PropertyList)	P		✓
10	getCount() : Int32 (in PropertyList)	P		✓
11	getConditionCount() : Int32 (in Rule)	P		✓
12	getConsequenceCount() : Int32 (in Rule)	P	✓	✓
13	addActionLevel(String id,Int32 level) : Boolean (in ActionsTestFixture)	T	✓	✓
14	addConditionOperationOperandAOperandB(String property,Operand operation,Int32 opA,Int32 opB) : void (in ActionsTestFixture)	T		✓
15	addPropertyValue(String attribute,Int32 value) : void (in ActionsTestFixture)	T		✓
16	changePropertyValue(String attribute,Int32 value) : void (in ActionsTestFixture)	T		✓
17	createMacro(String description) : void (in ActionsTestFixture)	T	✓	✓
18	deviceStatus(Int32 id) : Int32 (in ActionsTestFixture)	T	✓	✓
19	executeActions() : void (in ActionsTestFixture)	T	✓	
20	executeConditionalActions() : void (in ActionsTestFixture)	T		✓
21	numberOfActions() : Int32 (in ActionsTestFixture)	T	✓	✓
22	numberOfConditions() : Int32 (in ActionsTestFixture)	T		✓
23	numberOfProperties() : Int32 (in ActionsTestFixture)	T		✓
24	setupTwoDevices() : void (in ActionsTestFixture)	T	✓	✓

Table 4. Production code units for variant 1

Class	Methods
Property	No class-specific methods
~~Condition~~	~~isTrue() : Boolean~~
Rule	getConsequenceAt(Int32 index) : Property ~~addCondition(Condition condition) : Boolean~~ addConsequence(String deviceID,Int32 level) : Boolean ~~isSatisfiedBy(PropertyList properties) : Boolean~~ ~~getConditionCount() : Int32~~ getConsequenceCount() : Int32
~~PropertyList~~	~~getProperty(String attribute) : Property~~ ~~addProperty(Property property) : Boolean~~ ~~hasAttribute(String attribute) : Int32~~ ~~getCount() : Int32~~
DeviceList	getDeviceWhereID(Int32 id) : Device

```
namespace TestingPro
{
    public partial class ActionsTestFixture : DoFixture
    {
        DeviceList deviceList;
        Rule rule;
        PropertyList propList;
        public ActionsTestFixture(){...}
        public void setupTwoDevices(){...}
        public void createMacro(string description){...}
        public bool addActionLevel(string id, int level){...}
        public int deviceStatus(int id){...}
        public void executeActions(){...}
        public void executeConditionalActions(){...}
        public void addPropertyValue(string attribute, int value){...}
        public void changePropertyValue(string attribute, int value){...}
        public int numberOfActions(){...}
        public int numberOfProperties(){...}
        public int numberOfConditions(){...}
        public void addConditionOperationOperandAOperandB(string property,
                                Operand operation, int opA, int opB){...}
    }
```

Figure 9. Fixture code for variant 1

classes. We anticipate facing more challenges when we start dealing with UI classes, communication interfaces and hardware layers. This is because the testing tools currently available do not automate tests for these layers as effectively as they do for model classes. Resolving this issue is one of the tasks on our to-do list for this year.

4. Related Work

The quest for an incremental, non-invasive approach to establishing and managing production lines is a relatively new phenomenon. Kruger [10] built a

commercial tool to ease the transition to software mass customization. The tool utilizes the concept of separation of concerns to manage variability in software systems. Moreover, Clegg et al. [11] proposed a method to incrementally build an SPL architecture in an object-orientated environment. The method does not discuss how to extract and communicate variability from the requirement engineering phase to the realization phase. O'Brien et al. [12] discussed the introduction of SPLs in big organizations based on a top-down mining approach for core assets. They assume the organization has already developed many applications in the domain. Combining ASD and SPLs has been briefly discussed in literature. McGregor [13] presented an interesting theoretical attempt to reconstruct a hybrid method. Carbon et al. [9] proposed the use of a reuse-centric application engineering process to combine agile methods and SPLs. The approach gives agile methods the role of tailoring a product for a specific customer during application engineering. Another effort was by Hanssen et al. [14] where SPLs were used at the strategic level and ASD was used at the medium-term project level.

While these efforts are interesting attempts to combine concepts from ASD and SPLs, their focus is different from the focus of our research. The work presented here is focused on a test-driven approach to introduce product line practices to small organizations in a non-disruptive manner (rather than only managing existing SPLs). To the best of our knowledge, this research focus is original and has not been previously discussed in literature. The use of a test-driven approach in SPLs was initially proposed by Ghanam et al. [15].

5. Conclusions

This paper contributed a novel lightweight approach to manage variability in software product lines. The approach enables agile organizations – especially those adopting XP practices – to instantiate various products from a core system. These products, although different, are treated and managed as a single system with variation points. Combining SPL and XP practices provides significant advantages for software practitioners. It does not only reduce the amount of rework and the cost of producing customized solutions, but also makes it feasible for agile organizations to target customers with diverse needs without having to disturb the agility of their practices.

The notion of variability management does usually require a radical change in the mindset in the organization as soon as they start to look at their products as a family. However, the proposed approach has the potential to substantially reduce the adoption barriers of a product line practice through its incremental and non-burdening nature. The approach utilizes test artifacts that are naturally produced in XP projects, and gives customer involvement a special treatment by enabling customers to pick from variants and contribute to the variability model when available variants are not satisfactory.

The approach presented in this paper was evaluated through a case study where it actually was employed to manage variability in an intelligent home system. The example illustrated that the approach is feasible and useful, but suffered limitations that are to be addressed in future research. We are presently in the process of combining results from this work with concepts like reuse management to magnify the advantages of adopting an agile product line.

6. References

[1] Clements, P., and Northrop, L., *Software Product Lines: Practice and Patterns*, Addison-Wesley, US, 2001.

[2] About Software Product Lines, http://www.sei.cmu.edu/productlines/about_pl.html, accessed Dec, 2008.

[3] Pohl, K., Böckle, G., and Linden, F., *SPLE: Foundations, Principles and Techniques*, Springer, Germany, 2005.

[4] FIT, http://fit.c2.com, accessed Nov, 2008.

[5] Fitnesse, http://www.fitnesse.org, accessed Dec, 2008.

[6] VersionOne, http://www.versionone.com/Resources/FeatureEstimation.asp, accessed Dec, 2008.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.

[8] D. Avison, F. Lau, M. Myers, P. Nielsen, Action research, Communications of the ACM 42 (1) (1999) 94–97.

[9] Carbon, R., Lindvall, M., Muthig, D., and Costa, P. Integrating PL Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design, 1st International Workshop on Agile Product Line Engineering, 2006.

[10] Kruger, C., "Easing the Transition to Software Mass Customization", in *Proceedings of the 4th International Workshop on Product Family Engineering*, Germany, 2002.

[11] Clegg, K., Kelly, T., and McDermid, J., Incremental Product-Line Development, *International Workshop on Product Line Engineering*, Seattle, 2002.

[12] O'Brien, L., and Smith, D., MAP and OAR Methods: Techniques for Developing Core Assets for Software Product Lines from Existing Assets, *CMU/SEI-2002-TN-007*, 2002.

[13] McGregor, J. Agile Software Product Lines, Deconstructed, *Journal of Object Technology*, 7(8), 2008.

[14] Hanssen, G., and Fægri, T., "Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering", *Journal of Systems and Software*, 2008.

[15] Ghanam, Y., Park, S., and Maurer, F. A Test-Driven Approach to Establishing & Managing Agile Product Lines. The 5th SPLiT Workshop –SPLC 2008, Ireland.