The Role of Aspects in Modeling Product Line Variabilities

Jing (Janet) Liu
Department of Computer Science
Iowa State University
1-515-294-2735
janetlj@cs.iastate.edu

Robyn R. Lutz
Department of Computer Science
Iowa State University and
Jet Propulsion Laboratory/Caltech
1-515-294-3654
rlutz@cs.iastate.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
1-515-294-6168
hridesh@cs.iastate.edu

ABSTRACT

As of today, it is unclear whether aspect-oriented modeling can benefit the model-driven development of software product lines. Although some preliminary studies exist at the requirements and implementation level that investigate the interaction of crosscutting behaviors and product-line variabilities, to the best of our knowledge these interactions at the modeling level are not yet investigated. The contribution of this work is a preliminary study of the object-oriented and aspect-oriented approaches for handling crosscutting variabilities. This study helps us identify desired characteristics of aspect-oriented modeling techniques for product lines. A pacemaker product line, extracted from the real industry case, serves as a running example to illustrate our findings.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *Representation.*

General Terms

Design, Standardization.

Keywords

Model-Driven Development, Software Product Lines, Variability, Aspect, Aspect-Oriented Modeling.

1. INTRODUCTION

Model-driven development (MDD) [29], [30], [37] has played a very important role in software product-line engineering [18], [20], [26]. The executable models help exemplify the requirements, detect design flaws, validate the effects of variability management and help future maintenance [31]. However, the variability realization techniques in this area are geared toward local variabilities [7], [18]. We define crosscutting variabilities as those whose realizations are "fragmented across a system" [36]. We define local variabilities as those that can be captured in a modularized, object-oriented software artifact (e.g., a use case, an architectural block, a class, etc.) in the dominant decomposition. The lack of designated mechanisms for handling crosscutting variabilities in the product-line modeling level has hindered the sufficient support of all types of variabilities in MDD and created a void in validating design decisions regarding them.

Aspect-Oriented Software Development (AOSD) [15], [21], [23] has emerged as a promising solution for handling crosscutting concerns [47] in all phases of the software development lifecycle. Several approaches [5], [24], [25] have already extended AOSD into product-line requirements and implementation. Thus, it is natural to seek to combine Aspect-Oriented approaches with MDD in software product line practice.

This paper conducts a preliminary study of the Object-Oriented (OO) and Aspect-Oriented (AO) approaches in handling crosscutting, behavioral variabilities. A pacemaker product line, extracted from a real industry case, is used to illustrate our findings. For example, we observe that in this product line the AO approach handles variabilities that have a common mechanism but differ in locations better than the OO approach, if an automatic weaving mechanism is provided. However, the AO approach does not necessarily support more variability than the OO approach.

The rest of this paper is organized as follows. Section 2 presents needed background information. Section 3 introduces the running example (a pacemaker product line) and reports experience in modeling the crosscutting behavior using OO and AO techniques. Section 4 discusses our observations to some open problems found during the case study. Section 5 provides related work. Finally, Section 6 concludes and describes future work.

2. BACKGROUND

Model Driven Development (MDD) [29], [30], [37] is a software development approach that uses diagrams to communicate and uses models to understand and validate the designs, as well as to help software implementation, deployment and maintenance. It often uses the Object-Oriented paradigm [28] to abstract the system functionality into models.

A software product line is a set of software systems developed by a single company that share a common set of core requirements yet differ amongst each other according to a set of allowable variations [12], [49]. The product-line engineering concept is advantageous in that it exploits the potential for reusability in the analysis and development of the core and variable requirements in each member of the product line. Variability is the part of the software artifact that makes a product line member differ from others [49]. Four main approaches used to model variability in product lines are [48]: parameterization, information hiding, inheritance, and variation points. These approaches can be readily integrated with component engineering [34] and MDD [18]. However, none of them is designed to address those variabilities that crosscut multiple software artifacts. (See Sect. 3 for an example.)

AOSD [15], [21], [23] is emerging as a way to complement the traditional Object-Oriented Software Development by modularizing crosscutting concerns in a new software artifact called an aspect [23]. The places where an aspect crosscuts a software system are called join points [23].

Existing work in this area has covered a broad spectrum of the software development processes for single systems, from requirements analysis [8], [11], [25], [32], [38], architectural design [22], [40], modeling [1], [6], [45], coding [5], [24], and testing [50], [51]. Because of its ability in handling crosscutting concerns, AOSD is a natural candidate to manage crosscutting variabilities in a product line setting.

3. CASE STUDY

In this section we first present the running example, then give some concrete crosscutting variabilities and describe the modeling process using the OO and the AO approaches. Some findings are provided at the end.

3.1 Pacemaker Product Line

We use a pacemaker product line to evaluate different techniques for modeling crosscutting variabilities. These are real-time, embedded, and safety critical systems, that have been successfully developed in industry using MDD and software product line practices [26]. Studying the modeling techniques used for its variabilities can not only help enhance the safety assurance level of such systems, but may also yield observations that raise our confidence in other similar systems.

A pacemaker is an embedded medical device designed to monitor and regulate the beating of the heart when it is not beating at a normal rate. It consists of a monitoring device embedded in the chest area as well as a set of pacing leads (wires) from the monitoring device into the chambers of the heart [14]. In our simplified example, the monitoring device has three basic parts: a sensing part (sensor) that senses heart beat, a stimulation part (pulse generator) that generates pulses to the heart, and a controlling part (controller) that configures different pacing and sensing algorithms and issues commands.

In this example, we only consider a single-chambered product line of pacemakers that does pacing and sensing in the heart's ventricles. More advanced pacemakers can be dual-chamber, and the pacing or sensing algorithms applied to each chamber can be different although highly coordinated. In our case study, we consider three different products within this product line:

BasePapcemaker: A BasePacemaker has the basic functionality shared by all pacemakers: generating a pulse whenever no heart beat is sensed during the sensing interval.

ModeTransitivePacemaker: A ModeTransitivePacemaker can switch between Inhibited Mode and Triggered Mode during runtime. In the Inhibited Mode, the pacemaker acts exactly like a BasePacemaker. In the Triggered Mode, a pulse follows every heartbeat. (Triggered Mode is mainly used in therapies for dual-chamber pacemakers.)

RateResponsivePacemaker: A RateResponsivePacemaker acts similarly to the BasePacemaker but can adjust its sensing interval according to the patient’s current activity level: LRLrate, denoting the Lower Rate level for a patient’s normal activities and URL rate, denoting the Upper Rate Level when a patient is exercising.

3.2 An Example of Crosscutting Variability

Many of the major components in a pacemaker have to log their critical events into an EventRecorder component for use in making therapy decisions either by the pacemaker or by the doctors [14]. However, different pacemakers log different events at different relative or absolute times. Event Logging is a crosscutting variability whose functionality is shared among different components in each pacemaker system. Requirements and features for this product line are specified in [27] using a Commonality and Variability Analysis (CVA), as part of the FAST approach [49]. The excerption of CVA for the event logging is presented in Table 1. The variabilities and commonalities are detailed in Table 2.

Table 1. Excerpts from pacemaker product line Commonality & Variability Analysis

Commonality 1. A pacemaker shall log average heart rate sensed every fixed recording interval at the BaseSensor component.
Commonality 2. A pacemaker operating in Inhibited mode shall record the pulse width of every pulse being generated at the PulseGenerator component.
Variability 1. A pacemaker operating in Triggered mode shall record the average number of pulses generated every fixed recording interval at the PulseGenerator component.
Variability 2. A pacemaker with an extra sensor shall record the percentage of the pacemaker sensing at LRLrate every fixed recording interval at the ExtraSensor component.

Table 2. Event Logging Variability & Commonality

Product Name	Component Name	Events to Log
Base Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
Mode Transitive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	1) In the Triggered mode, the average number of pulses generated every fixed recording interval 2) In the Inhibited mode, the pulse width of every pulse being generated
Rate Responsive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
	Extra Sensor	The percentage of the pacemaker sensing at LRLrate every fixed recording interval

3.3 Modeling using OO techniques

The Object Management Group (OMG) [33] uses UML [10] as a standard language for the Model-Driven Architecture [30]. In this section, we are using the UML 2.0 statechart [10] to model crosscutting variabilities. Statechart was preferred over other modeling artifacts for two reasons. First, it is particular suitable for detailed behavioral modeling. Second, it is close to implementation and is crucial in generating executable models to validate the design. The successful modeling in statecharts not only guides the implementation, but also provides assurance for later stages.

The following subsections describe the process of incremental modeling [27] of the crosscutting variabilities in different products. It is supported by the Rhapsody software modeling environment [13] from I-Logix. We start from the product that has the fewest variations (i.e., the BasePacemaker), and then incrementally build the model with variations of other products in the product line.

3.3.1 *BasePacemaker*

Based on the UML statechart model for the pacemaker product lines described in our previous work [27], we add the behavior of the EventRecorder of the BasePacemaker using the statechart shown in Fig. 1. It is composed of three orthogonal statecharts [10]: the BaseSensorCounting and BaseSensorRecording subcharts for recording the average heart rate at every recordingInterval, and the PulseGeneratorRecording subchart for recording the pulse width every time a pulse is generated.

In order to get the pulse width value (denoting how long the pulse lasts), which is a private attribute of the PulseGenerator Class, the PulseGenerator has to send this value explicitly as a parameter of the evPulseDone message (Fig. 2). The "show(params->width)" in Fig. 1 is a function that prints the value of the parameter named "width" (which is the parameter of "evPulseDone").

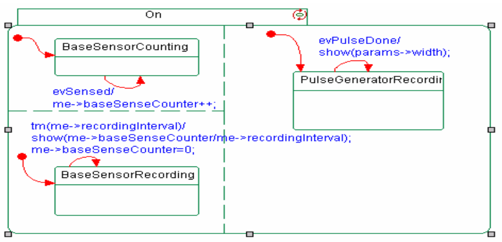

Figure 1. BasePacemaker's EventRecorder

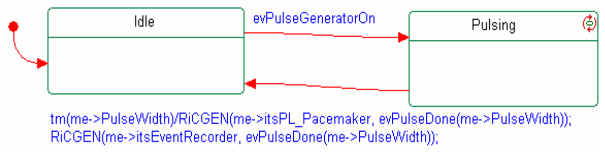
Idle
evPulseGeneratorOn
Pulsing
tm(me->PulseWidth)/RiCGEN(me->itsPL_Pacemaker, evPulseDone(me->PulseWidth));
RiCGEN(me->itsEventRecorder, evPulseDone(me->PulseWidth));

Figure 2. BasePacemaker's PulseGenerator

3.3.2 *ModeTransitivePacemaker*

The statechart of EventRecorder in the ModeTransitivePacemaker, shown in Fig. 3, is created by inheriting [18] the EventRecorder's statechart from the BasePacemaker. Variability 1 in Table 1 (mode transitive) is modeled by adding a condition connector [10] (the symbol of a circle with a "C" inside) in the sub-chart for pulse recording, and by adding a new subchart of pulse counting. The sub-chart of mode transitions (InhibitedMode and TriggeredMode) is created due to the need to keep the mode attributes local (as a private member, required by the modeling tool Rhapsody [13], as well as a common practice in Object- Oriented software development).

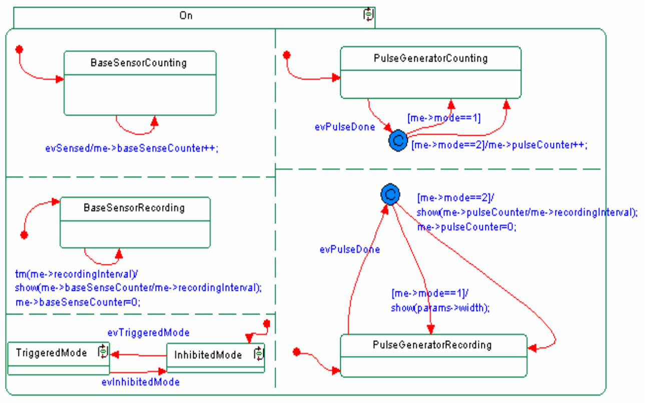

Figure 3. ModeTransitivePacemaker's EventRecorder

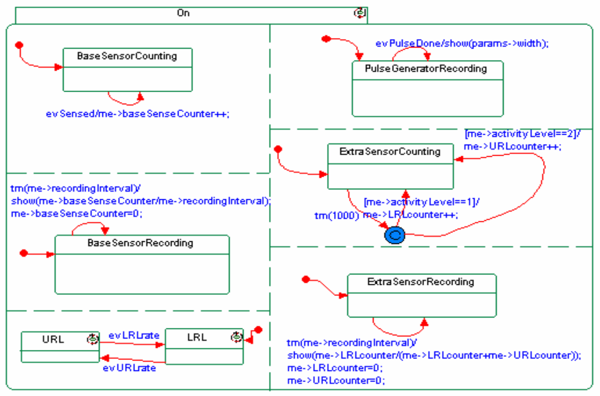

Figure 4. RateResponsivePacemaker's EventRecorder

3.3.3 *RateResponsivePacemaker*

There are two ways to implement the statechart for the EventRecorder in the RateResponsivePacemaker. The first is to create an EventRecorder statechart for the whole product line (we call it PL_EventRecorder) by introducing the Variability 2 in Table 1 (the rate responsive variability) into the EventRecorder statechart of ModeTransitivePacemaker via transitions with condition connectors. This way the PL_EventRecorder becomes a parameterized state model [18] for the whole product line. This method is described in detail in our previous work [27]. The second way is to inherit the statechart of EventRecorder in BasePacemaker. As a result, each product member has its own statechart deriving from a base statechart (the BasePacemaker's). These two ways are the common choices in modeling variabilities using statecharts in a software product line [18]. For ease of illustration of the variability we show the statechart generated using the second method in Fig. 4.

As seen in Fig. 4, Variability 2 in Table 1 is modeled by adding a sub-chart for ExtraSensor counting and recording separately. As in the ModeTransitivePacemaker, a sub-chart of activity level (URL and LRL) is created.

Thus, in the OO approach, the EventRecorder component acts similarly to an Observer Pattern [16]: it monitors all the triggering events and then dispatches them to their separate handlers (orthogonal sub-charts).

3.4 Modeling using AO techniques

Due to the lack of standard AO modeling techniques and support for weaving mechanism, we use UML sequence diagrams together with textual descriptions to demonstrate the behavior of an aspect. Sequence diagrams [10] capture the dynamic view of a system. They show a set of roles and the messages that are passed between instances of the roles. Sequence diagrams have been used before to demonstrate the behavior of aspects [6], [11], [43]. In this case, the sequence diagram serves as an abstraction to demonstrate the characteristics of common AO techniques.

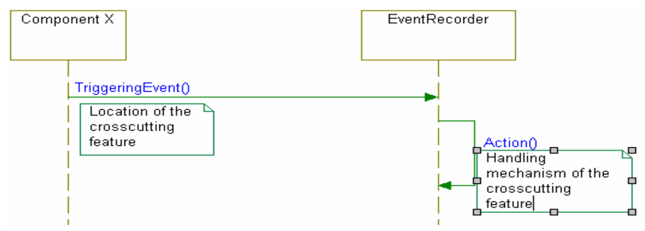
Component X
EventRecorder
TriggeringEvent()
Location of the crosscutting feature
Action()
Handling mechanism of the crosscutting feature

Figure 5. Generic Scenario of the EventRecorder Aspect

The EventRecorder component in our example system encapsulates the crosscutting variability of event logging. Therefore, we choose to model this component as an aspect. The generic scenario of the EventRecorder aspect is depicted in Fig. 5. It is composed of two parts: the *triggering event*, which is the location where the aspect crosscuts (call it "*location*"), and the *action*, which is the behavior of the aspect after being triggered (call it "*mechanism*"). Table 3 illustrates the different locations and mechanisms for the EventRecorder aspect in the product line (the events and action names are the abstraction of their counterparts in Fig. 1, 2 and 3). Table 3 shows that several locations share similar mechanisms. Table 4 helps demonstrate this in a clearer fashion.

We make the following observations by comparing Table 3 and Table 4:

1) Each group of locations that share a similar mechanism can be modeled as a "pointcut" [23], while the similar mechanism can be modeled as an "advice" [23]. By "similar" we mean that they behave the same except for the context to which they apply. For example, the counter incrementing behavior in different components is similar, except for the variable it increments.

In some cases, mechanisms differ significantly at different locations. For example, the mechanism for the location "RateResponsivePacemaker -> ExtraSensor -> recording interval timeout" differs from the second mechanism in Table 4 because the first takes the sum and the second takes the average. These mechanisms cannot be modeled as a single advice.

2) Here, where there is only a single crosscutting variability, the mechanisms do not overlap. This is because, even if two mechanisms apply to the same locations, they happen under different conditions. Thus, it does not make much difference whether we model each of the matching pointcut and advice pairs (as described above) in a separate aspect or model all of them in one aspect.

Table 3. Aspect Specification

Product Name	**Component Name**	**Aspect**	
		Join Point	**Advice**
Base Pacemaker	Base Sensor	Sensed	counter increases by one
		recording interval timeout	record the average counter value during the recording interval, then reset the counter
	Pulse Generator	Pulse	record the pulse width
Mode Transitive Pacemaker	Base Sensor	Same as in BasePacemaker	
	Pulse Generator	Pulse	1) if in Inhibited mode, same as BasePacemaker 2) if in Triggered mode, counter increases by one
		recording interval timeout	1) if in Inhibited mode, do nothing 2) if in Triggered mode, record the average counter value during the recording interval, then reset the counter
Rate Responsive Pacemaker	Base Sensor	Same as in BasePacemaker	
	Pulse Generator	Same as in BasePacemaker	
	Extra Sensor	1 msec timeout	1) if in LRLrate, LRLrate counter increases by one 2) if in URLrate, URLrate counter increases by one
		recording interval timeout	Record the ratio of the LRLrate counter value to the sum of the LRLrate and URLrate counter values, then reset the counters

Table 4. Mechanism Classification

Mechanism	Location	Condition
Counter residing in the same component as the location increases by one	1) BasePacemaker->BaseSensor->sensed event 2) ModeTransitivePacemaker->BaseSensor->sensed event 3) ModeTransitivePacemaker->PulseGenerator->sensed event 4) RateResponsivePacemaker -> BaseSensor ->sensed event 5) RateResponsivePacemaker -> ExtraSensor -> 1 msec timeout	3) if in Inhibited Mode 5) if in LRLrate, increase LRLrate counter; if in URLrate, increase URLrate counter
Record the average counter value during the recording interval, then reset the counter	1) BasePacemaker -> BaseSensor -> recording interval timeout 2) ModeTransitivePacemaker -> PulseGenerator -> recording interval timeout 3) RateResponsivePacemaker ->BaseSensor -> recording interval timeout	2) if in Triggered Mode
Record the pulse width	1) BasePacemaker -> PulseGenerator -> pulse event 2) ModeTransitivePacemaker -> PulseGenerator -> pulse event 3) ModeTransitivePacemaker -> PulseGenerator -> pulse event	2) if in Triggered Mode

However, if we introduce another crosscutting variability into the product line, it is likely that the mechanisms from the two variabilities will overlap in locations. In that case, conflict resolving techniques are needed. These could be similar to the feature interaction handling mechanisms [35] for local variabilities, but we have to bear in mind that such conflict resolution will apply invasively in the AO setting (rather than locally as in the OO setting). In fact, the tool support for aspect interaction at the programming level [3], [39] may be migrated to the modeling level.

3) The locations to which a crosscutting variability applies to can be fragmented within and across a product. For example, locations that share a similar mechanism can reside in different components of the same product, or in components from different products. This means that the scope of the join point (as well as the weaving) needs to be extended to the product-line level, rather than the product level as in traditional AOSD.

4) There are two ways that a condition can affect the mechanism. In the first way (seen in the first condition in the first mechanism group in Table 4) the condition serves as a *switch* to decide whether an event is able to trigger the action. In the second way (seen in the second condition in the same group) the condition uses *context* information passed to tell where the action should apply. Consequently, these two types of conditions need to be modeled differently. This remains an open problem for our future work.

3.5 Findings

Some similarities and differences between the OO approach and AO approach are observed as follows:

1. Both approaches handle the crosscutting variability in a centralized manner. The OO approach invokes the methods explicitly while the AO approach handles it implicitly [17], [44].

2. The OO approach requires each component being monitored to send its local variable values explicitly via messages, since the local variables are private in the OO paradigm. However, in the AO approach, the aspects are allowed reflective access to certain variables at the join points, such as the executing object, the target of a call, arguments of a method, etc. Explicitly sending these variables is not necessary in the AO approach.

3. In the OO approach, the location where the handling mechanism takes place (after the triggering event) must involve a component other than the component that sends the triggering event. However in the AO approach, there is no such restriction. This is due to the similar reason as above.

4. In the AO approach, if we treat different locations that share a similar mechanism as join points for the same aspect, modeling variabilities that have a common mechanism but differ in locations will be easier than in the OO approach, assuming automatic weaving mechanisms are provided. This is because in the OO approach, users have to manually adapt the variability into the local context, while in the AO approach users simply need to add some new join points. This is true for variabilities both within a product and across several products. In this situation, the AO approach makes the modeling of crosscutting features more reusable across the software product line.

5. The AO approach does not support more variability than the OO approach, since each different handling mechanism requires a separate advice. With many variations in the handling mechanism, both the AO approach and the OO approach incur significant overhead. Creating aspect templates or generic aspects helps reuse, but does not accommodate more variabilities.

4. DISCUSSION

In this section we give some suggestions for the weaving mechanism in the modeling level, as well as two open problems confronted in this work. Finally, a set of criteria for future empirical studies is proposed.

4.1 Weaving Mechanism

Without concrete weaving mechanisms, no executable models can be generated from the AO modeling. Weaving at the modeling level also provides a way to generate models independent of implementation languages. Based on our experiences using Rhapsody [13] as an OO modeling tool, we propose some suggestions for the weaving mechanism at the modeling level.

1. The effect of the aspect should be able to be demonstrated in the animation of the executable model. In other words, users should be able to model the aspect and the rest of the system separately and see the effect of weaving in the animation.

2. Users should be able to choose to implement the aspect weaver themselves by building it in the models, or to choose an existing weaver. For the latter, users should be able to turn it on or off.

3. Users should be able to view the marked join point, attributes and methods (advices) introduced by aspects statically in the system model, even though they cannot use them other than in the aspect.

4.2 Open Problems

The first open problem is about whether to model local features (e.g., switching to Inhibited Mode during runtime) at the product-line level using the AO approach. As suggested in Section 3.5, if we extend the scope of weaving and join point to the product line level, e.g., advising several product members using one aspect, we can achieve greater reuse of the crosscutting features. That raises the question of whether we can and should do the same for those local features. Some preliminary case studies can be found at [2] and [4].

The second problem is how much obliviousness a modeler should have about the weaving process. Unlike the coding stage, the modeling stage calls for exemplification of the design intent. Therefore we expect more knowledge about the weaving process to be exposed in the modeling level than in the AOP level. However, how much is enough remains a problem for future research.

4.3 Evaluation Criteria

In this section we propose the criteria for comparing the capability of different modeling techniques for crosscutting variabilities. The metrics introduced here, while preliminary and partial, identify some criteria that may be useful in subsequent, more empirical evaluations.

Feasibility

This criterion evaluates if it is easy or possible to model all types of crosscutting variabilities. In order to do this, a taxonomy of crosscutting variabilities needs to be provided. Anastasopoulos and Muthig [3] have done an initial step by classifying variations into two types: "positive" and "negative", denoting the effect of variability on the system (i.e., adding vs. removing functionalities).

Degrees of variability

This denotes how flexible the modeling technique is for modeling the variability. Note that the OO and AO approaches can have different notions of "flexibility". For instance, in the OO approach, binding time [46] is used to describe how late developers are able to change a variability (or select a variant at a specific variation point). However, this notion is not very meaningful for the AO approach as most aspects are bound at compilation time and the rest at load time or run time.

Therefore, we propose to measure the degree of crosscutting variability by evaluating the limitation of mechanisms and diversity of locations where variability can occur. (Point 4 of Section 3.5 provided such an example.)

Evolution

This is an important issue in software product lines. Specifically, we need to evaluate if an approach supports changing requirements and the addition of new product-line members. This can be done by checking the likely impact introduced by a change.

Executable model

As stated at the beginning of this paper, executable models are very important in clarifying the design intent and validating design logics. This is an indispensable part in MDD. We examine this criterion by checking if the modeling language provides sufficient support for describing behaviors and if code generation (for both the system and environment) is available.

Tool support

Tool support is crucial in making an approach scalable, especially in a product line setting. With sufficient tool support, the code generation should be automatically done. Moreover, users should be able to run the executable model and check it against the requirements scenarios [13].

Cost

Just as in product-line engineering, where too few products do not provide a gain via reuse [49], a new modeling technique for the crosscutting features does not necessarily always save time and money. We need to identify the situations when it will receive the biggest gain and maybe provide a pay-off model for such a technique.

5. RELATED WORK

Existing work that introduced the concept of aspects into software product-line development include [3], [5], [9], [19], [24], [25], and [39].

The work by Apel et. al. [5] combines the force of Feature Oriented Programming (FOP) and Aspect Oriented Programming (AOP) in the code level. Loughran and Rashid [24] propose 'framed aspects' as a technique combining AOP, frame technology and Feature-Oriented Domain Analysis (FODA). Both [5] and [24] compare the aspect-oriented approach with the approach they propose to combine with (mixin layers and frame respectively) and conclude that they complement each other. This also backs up our findings that OO and AO variability modeling techniques complement each other, such as AOP and OOP.

Anastasopoulos and Muthig [3], as well as Saleh and Gomaa [39], present evaluations of the use of AOP in the implementation of software product lines. Concrete tool support is provided for automatic weaving [39] or configuration [3].

Griss [19] proposes a feature-driven analysis to find aspects as crosscutting features at the high level and map them into code fragments in the components in the low level. The feature analysis provides the traceability document through the development cycle.

Loughran et. al. [25] introduce NAPLES, a tool that uses natural language processing and aspect-oriented techniques to derive feature-oriented models (including features, aspects, variabilities and commonalities in a given domain) from requirements.

Batory et. al. [9] models the components of distributed simulations as aspects, via the help of DSLs and GenVoca PLAs.

A significant amount of work has been devoted to aspect-oriented modeling for single systems, e.g., [6], [21], [41], [42], [43], and [51].

However, none of the above work addresses the role of aspects in the model-driven development of product lines in contrast to the traditional OO approach, as we do here.

6. CONCLUSION

The work described here provides a preliminary comparison of the OO and AO approaches in modeling crosscutting variabilities, based on experience with a product line case study. Several observations are made that may be helpful for future research. Possible future work includes tools for resolving aspect conflicts, more empirical evaluations of the two approaches, a rigid weaving mechanism, and its implementation in an existing MDD tool.

7. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants 0204139, 0205588 and 0541163.

8. REFERENCES

[1] Aldawud, O., Bader, A., and Elrad T. Weaving with Statecharts. in *the Aspect-Oriented Modeling with UML workshop at the 1st Int'l Conf. on Aspect-Oriented Software Development* (Enschede, The Netherlands, 2002).

[2] Alves, V., Matos, P. Jr., and Borba, P. An Incremental Aspect-Oriented Product Line Method for J2ME Game Development *in the Workshop on Managing Variabilities Consistently in Design and Code at the 19th OOPSLA*, (Vancouver, Canada, 2004).

[3] Anastasopoulos, M., and Muthig, D., An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. in *Software Reuse: Methods, Techniques and Tools: 8th Int'l Conf., ICSR 2004* (Madrid, Spain, 2004), Springer Berlin / Heidelberg, 141-156.

[4] Apel, S., and Batory, D.,When to Use Features and Aspects? A Case Study. In *Proc. GPCE 2006* (Portland, USA, 2006).

[5] Apel, S., Leich, T., and Saake, G., Aspectual Mixin Layers: Aspects and Features in Concert. in *the 27th ICSE.* (Shanghai, China, 2006), ACM Press, 122 – 131.

[6] Araújo, J., Whittle, J., and Kim, D. Modeling and Composing Scenario-based Requirements with Aspects. in *the 12th IEEE International Requirements Engineering Conference* (Kyoto, Japan, 2004), IEEE Press, 58-67.

[7] Atkinson, C. et. al. *Component-based Product Line Engineering with UML*. Addison-Wesley Professional, 2001.

[8] Baniassad, E. et. al. Discovering Early Aspects. *IEEE Software, 23,* 1 (Jan. 2006), 61-70.

[9] Batory, D., et. al.. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology, 11*, 2 (April 2002), 191-214.

[10] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005.

[11] Clarke, S., and Baniassad, E. *Aspect-oriented Analysis and Design: The Theme Approach*, Addison-Wesley, Upper Saddle River, 2005.

[12] Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] Douglass, B. P. *Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.* Addison-Wesley, 1999.

[14] Ellenbogen, K.A., and Wood M.A. *Cardiac Pacing and ICDs*. Blackwell Publishing, Malden, 2005.

[15] Filman R. E., Elrad, T., Clarke, S., and Aksit, M. et. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.

[16] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[17] Garlan, D., and Notkin, D. Formalizing Design Spaces: Implicit Invocation Mechanisms. in *VDM '91: Formal Software Development Methods*, (Noordwijkerhout, The Netherlands, 1991), Springer-Verlag, 31-44.

[18] Gomaa, H. *Designing Software Product Lines with UML: From Uses Cases to Pattern-Based Software Architectures.* Addison-Wesley, Boston, 2005.

[19] Griss, M. L., Implementing Product-line Features By Composing Component Aspects. in the *First International Software Product Line Conference* (Denver, USA, 2000). Kluwer 2000, 271-289.

[20] Guidant Corporation Keeps Its Rhythm With Statement MAGNUM, I-Logix, 2002. Retrieved August 7, 2006, from Iowa State University: http://www.ilogix.com/pdf/success/ Statemate_GuidantCorporationKeepsItsRhythm.pdf.

[21] Jacobson, I., and Ng, P. *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley Professional, Upper Saddle River, 2004.

[22] Katara, M., and Katz, S. Architectural Views of Aspects. in the 2nd *Int'l Conf. on Aspect-oriented Software Development* (Boston, USA, 2003), ACM Press, 1-10.

[23] Kiczales, G. et. al., Aspect-Oriented Programming. in *the 11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland, 1997), Springer-Verlag, 220-242.

[24] Loughran, N., and Rashid, A., Framed Aspects: Supporting Variability and Configurability for AOP. *in the 8th International Conference on Software Reuse* (Madrid, Spain, 2004), Springer, 127-140.

[25] Loughran, N., Sampaio, A., and Rashid A., From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. *MoDELS 2005 International Workshop on MDD in Product Lines* (Montego Bay, Jamaica, 2005), Springer, 262-271.

[26] Liu, J., Lutz, R., and Thompson J, Mapping Concern Space to Software Architecture: A Connector-Based Approach. in

ICSE 2005 Workshop on Modeling and Analysis of Concerns in Software (St. Louis, USA, 2005), ACM SIGSOFT Software Engineering Notes (Volume 30, Issue 4), 1 – 5.

[27] Liu, J., Dehlinger, J., and Lutz, R. Safety Analysis of Software Product Lines using State-based Modeling. in *the 16th IEEE International Symposium on Software Reliability Engineering* (Chicago, USA, 2005), IEEE Press, 21-35.

[28] McgGregor, J. and Korson, T. Understanding Object-Oriented: A Unifying Paradigm. *Communication of the ACM, 33,* 9 (Sept. 1990), 40-60.

[29] Model-Driven Software Development, May 2006. Retrieved August 7, 2006, from Iowa State University: http://www.mdsd.info/mdsd_cm/page.php?page=intro&id=5.

[30] Mukerji, J., and Miller, J. The MDA Guide v1.0.1. *OMG Papers on the MDA,* June 2003. Retrieved August 7, 2006, from Iowa State University: http://www.omg.org/docs/omg/03-06-01.pdf.

[31] Niemann, S. Executable Systems Design with UML 2.0. *OMG Whitepapers on UML*, I-Logix, August 2004. Retrieved August 7, 2006, from Iowa State University: http://www.omg.org/news/whitepapers/ Executable_System_Design_UML.pdf

[32] Nuseibeh, B., Crosscutting Requirements. in *the 3rd International Conference on Aspect-oriented Software Development* (Lancaster, UK, 2004), ACM Press, 3-4.

[33] The Object Management Group (OMG), August 2006. Retrieved August 8, 2006, from Iowa State University: http://www.omg.org/.

[34] Pohl, K., Böckle, G., and van der Linden, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin, 2005.

[35] Prehofer, C. An Object-Oriented Approach to Feature Interaction. in *the 4th IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems* (Montréal, Canada, 1997), IOS Press, 313-325.

[36] Rajan, H. and Sullivan, K, Classpects: Unifying Aspect- and Object-Oriented Language Design. *in the 27th International Conference on Software Engineering*1(St. Louis, USA, 2005), The ACM Digital Library, 59 – 68.

[37] Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer 39,* 2 (Feb. 2006), 25-31.

[38] Rashid, A. et. al. Modularization and Composition of Aspectual Requirements. in the 2nd *International Conference on Aspect-oriented Software Development* (Boston, USA, 2003), ACM Press, 11-20.

[39] Saleh, M., and Gomaa, H., Separation of concerns in software product line engineering. in *ICSE 2005 Workshop on Modeling and Analysis of Concerns in Software* (St. Louis, USA, 2005), ACM SIGSOFT Software Engineering Notes (Volume 30, Issue 4), 1 – 5.

[40] Shomrat, M., and Yehudai, A. Obvious or not? Regulating architectural decisions using aspect-oriented programming. in the 1st *International Conference on Aspect-oriented Software Development* (Enschede, The Netherlands, 2002), ACM Press, 3-9.

[41] Sillito, J., Dutchyn, C., Eisenberg, A., and K. DeVolder. Use case level pointcuts. In *Proc. ECOOP 2004,* (Oslo, Norway, 2004).

[42] Stein, D., Hanenberg, S., and Unland, R., Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features. in *the 3rd International Workshop on Aspect-Oriented Modeling* (Boston, USA, 2003).

[43] Stein, D., Hanenberg, S., and Unland, R., On Representing Join Points in the UML. in *the 2nd International Workshop on Aspect-Oriented Modeling with UML* (Dresden, Germany, 2002).

[44] Sullivan, K., and Notkin, D. Reconciling Environment Integration and Software Evolution. *ACM Transaction on Software Engineering and Methodology, 1,* 3 (July 1992), 229-268.

[45] Sutton, S.M., and Rouvellou, I. Modeling of Software Concerns in Cosmos. in *the 1st International Conference on Aspect-oriented Software Development* (Enschede, The Netherlands, 2002), ACM Press, 127-133.

[46] Svahnberg, M., van Gurp, J., and Bosch, J. A taxonomy of variability realization techniques: Research Articles. *Software-Practice & Experience, 35,* 8 (July 2005), 705-754.

[47] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns. *in 21st Int'l Conf. on Software Engineering* (Los Angeles, USA, 1999), ACM Press, 107-119.

[48] Webber, D., and Gomaa, H. Modeling Variability in Software Product Lines with the Variation Point Model. *Science of Computer Programming, 53,* 3 (Dec. 2004), 305-331.

[49] Weiss, D. M., and Lai, C. T. R. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[50] Xie, T., and Zhao, J. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. in *the 5th International Conference on Aspect-oriented Software Development* (Bonn, Germany, 2006), ACM Press, 190-201.

[51] Xu, D., and Xu, W. State-based Incremental Testing of Aspect-oriented Programs. in *the 5th International Conference on Aspect-oriented Software Development* (Bonn, Germany, 2006), ACM Press, 180-189.