Categorizing and Modeling Variation in Families of Systems: a Position Paper

Borislava I. Simidchieva
bis@cs.umass.edu

Leon J. Osterweil
ljo@cs.umass.edu

Laboratory for Advanced Software Engineering Research (LASER)
Department of Computer Science
University of Massachusetts Amherst
140 Governors Drive, Amherst, MA 01003

ABSTRACT

This paper presents an approach that considers variation in systems and system architectures according to the kind of relation among the variants in the software family. The approach highlights why it is beneficial to consider such different variation relations separately and gives examples of what these relations may be.

Two main categories of variation relations are presented, based on whether the system architecture remains constant (*architecture-based variation*), or whether the architecture itself is variable, i.e. the variants do not share a common architecture. The paper introduces several different kinds of variation families that seem to belong to these two categories, as well as yet other families comprising variants that do not neatly fit in either category, with only a subset of the variants sharing a common architecture. Each kind of variation relation is illustrated with an example software family from different domains, including operating systems (OS).

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*description, interconnection, definition*; D.2.9 [**Software Engineering**]: Software Management—*software configuration management*

Keywords

software families, software product lines, system architectures, variation, variability

1. INTRODUCTION

A successful software development project rarely aims to produce a single piece of software. If the project is successful, then its product will be a capability that will need to run on many different platforms, is likely to present itself to different types of users in different ways, may need to run faster for some users, and have different sets of features for different kinds of users. In such cases it can be very ungainly to respond to differing requirements by building a single piece of software that is capable of meeting all of these needs. Instead these needs are typically met by building different *variants* of the software, with each of the variants being developed to meet different combinations of these needs. Increasingly complex and demanding requirements can be expected to cause the creation of ever larger sets of such variants, but it is important, nevertheless, that the different variants retain well-understood relations to each other. If such a well-understood relationship exists, the variants comprise what we refer to as a *software family*.

There are many reasons why it is important to have a firm understanding of how the variants in a software family are related to each other. If there are well-understood relations that define what variants are members of the software family (i.e. some collection of relations that define membership constraints and criteria), then it may be straightforward to create new variants as transformations of existing variants. Moreover, if there are formally and rigorously defined semantic relations that determine how variants are related, then it may be possible to safely conclude that newly created variants are guaranteed to have certain desirable properties. Once the relations among all variants in a software family are well defined and understood, then it may be possible to go a step further and make assurances about the entire collection of variants and whether the family itself has certain properties that would by construction be true of all its variants.

For these reasons there seems to be considerable value in identifying the kinds of relations that define families that have these sorts of desirable properties. The goal of this paper is to demonstrate that there are numerous kinds of variation relations that might be helpful in supporting such reasoning, and to indicate that different kinds of variation lead to families having different kinds of properties. As suggested above, some kinds of variation relations define families whose members differ only in speed. Other kinds of variation define families whose members offer functionality that differs only in relatively small, well-circumscribed ways. Still other kinds of variation relations define families whose members all achieve similar, or identical, goals, but do so in very different ways.

This paper identifies several kinds of variation that characterize families whose existence in the world is rather familiar and common, and describes them informally. We note that many of the different kinds of variation seem to arise directly from different components of a requirements specification. Often this need for variation is not stated explicitly, but can be readily identified as being implicit. We suggest that it is important for such variation requirements to be explicit, as this would seem to facilitate being explicit and precise about the variation relations that define the families that can respond successfully to these needs. Ideally the result should be a family in which the members are characterized formally and rigorously. Such rigorous characterizations are not presented in this paper, although the value in being able to do so is described and is suggested as an important goal for ongoing research.

While it may be quite challenging to define rigorously the variation relation that defines some important kinds of families, it appears that it might be relatively easy to define other kinds of families, especially when variability requirements are suitably clear and precisely stated. Initial attempts to characterize some kinds of software families have suggested that some of these characterizations might be based upon a shared architecture that underlies the family, where the different family members could be viewed as different implementations all based upon such a shared architecture. These kinds of variation are referred to as *architecture-based variation* in this paper.

However, the paper also identifies other kinds of software families that do not seem to be easily defined in terms of architecture-based variation, but rather arise from making modifications to underlying architectures. Some of the challenges and rewards of dealing with software families that fall in these two different categories of variation are outlined in the paper. The paper presents examples of families for which the variants share a common architecture and for which they do not. In addition that paper identifies other variant families that do not fit neatly into either category.

The rest of this paper is organized as follows. Section 2 enumerates some of the different kinds of variation the authors have observed, presenting examples from different domains of how each kind of variation relation may manifest. Section 3 revisits these different kinds of variation, and provides some discussion and intuition on how each kind of variation may be best managed and implemented. Sections 4 and 5 contain brief discussions of related work and future research directions, respectively. Finally, Section 6 presents some conclusions.

2. CATEGORIZING VARIATION

In this section, different kinds of variation are presented. What follows is not an exhaustive list but rather is intended to suggest that the range of kinds of software families is quite wide. Nevertheless, emphasis is placed upon how frequently families of variants seem to arise quite naturally in response to variation in different components of a requirements specification. A hypothetical software family of operating systems (OS) is used as a source for some of the examples to illustrate the different kinds of variation. But each subsection also provides other examples illustrating the specific kind of variation.

Terminology

The terminology used in this paper strives to follow conventional terminology used in the software product line engineering (SPLE) community. Specifically in this paper the term "software family" is used to denote a collection of software systems that are determined to be closely related based on a specific variation relation. Moreover, each member of a software family is called a "variant." Variants differ from one another in the functionality they provide, their speed, robustness, or any of a number of other aspects; this difference between variants is called "variation." When multiple variants all share some common features (e.g. sets of components, services, architecture specifications, or other artifacts), these commonalities will be referred to as "core assets." The process of determining what variants best fit together in a software family and how to most effectively design the family in order to maximize reuse is known as "commonality and variability analysis."

2.1 Functional Detail Variation

The members of a Functional Detail Variation family differ from each other in the amount of detail with which different functional capabilities are specified. Thus, different variants of a system may need to provide different levels of functionality based on the requirements for variation in the functionalities of different system variants.

For example, a high-end OS variant may have to satisfy a requirement for providing easy, elaborate built-in remote access so that its professional customers would have no problem accessing their files and applications when away from their computers, while a lower-end variant might be required only to provide more rudimentary functionality for remote access or less guidance on how to use such functionality.

Outside of the OS domain, an online web service may require a customer to provide certain information. But different variants of the service may be required to provide more assistance than others in assuring that the information provided is correct and adequate. Similarly, the requirements for a verification system might specify that an artifact is to be analyzed and certified as being suitable for use. But different variants of this system might specify the need for the application of different specific regimes by which this analysis and certification is to be done.

In all of these cases, the variants all support the provision of the same high-level functionality, but the requirements also mandate that different variants are needed. Some of these variants are required to offer additional functional capabilities, sometimes through implementations for which the specific details are specified.

It is reasonable to suggest that a Functional Detail Variation family might well consist entirely of members that share the same high level architecture. This seems reasonable as this kind of family is intended to respond to a fixed high level functional requirement specification. A key advantage of this is that the common high-level architecture might then be exploited as the basis for reasoning that could result in establishing assurances about the nature of the entire family. Under some circumstances it might be possible to be assured that an analysis of the common architecture provides results that hold for all family members. In other circumstances, it might be possible to build upon testing or analysis results applied to the common architecture to reduce (perhaps substantially) the effort required to verify a new member of the family. This could lead to an amortization of testing and analysis costs and faster development, with less risk of unknowingly introducing errors in variants.

2.2 Robustness Variation

The members of a Robustness Variation family differ from each other in the extent to which they are able to recover from incorrect or abusive use. Thus, for example, an operating system may simply crash when a user inputs an unknown command. Different variants of this operating system, however, might reply that the command is unknown and request that the user try again. Still other variants might try to prompt the user, might limit the number of retries, etc.

A high-end variant of the OS software family may boast automatic document backup so that automatic recovery over a local network is possible when certain failures have occurred. A lower-end variant may lack these capabilities, or may only provide document backup and recovery options to and from a local storage device and not a remote network server. Additionally, variants of the operating system targeting professional users may provide yet more sophisticated functionality assuring better system robustness, such as complete system backups, including user and system files and OS settings, in order to better protect users against hardware failures, for example.

Similarly, a safety-critical system such as a medication infusion pump might require that requests for medication dosage always fall within prescribed ranges. Some variants of the infusion pump software might simply refuse to provide any medication in response to an out-of-range request, while other variants might send diagnostic replies of varying degrees of thoroughness and utility, and still others might infuse some kind of default level of medication.

In all of these cases, the variants provide the same capabilities under nominal operating circumstances, but provide different kinds and levels of support when non-nominal conditions are encountered. Note that different subsets of the family will probably deal with different non-nominal situations, and different members of such a subset will presumably deal with the same non-nominal situations differently. Robustness variation tends to be driven by different robustness requirements that each variant needs to meet.

The Functional detail variation examples presented in section 2.1 were driven by requirements for variation in the nominal flow of events, rather than variation in exceptional flow. However, a rigorous, detailed, and precise specification of possible exceptions and how they are to be handled should also be an integral part of the specification of the requirements for any complicated software system. Such specifications of the need for different exceptional flow might well be met by a Robustness Variation family.

Robustness variation might seem intuitively to be quite different from functional detail variation. Although these two kinds of variation relations are clearly interrelated, they may be quite different from each other, depending upon how these two variation relations are formally defined. The examples of robustness variation we have described here might be included as a special case of functional detail variation instead of separately as robustness variation families. For example different variants from a functional detail variation family may produce identical robustness variation families, if the functional detail variation does not affect those points of the nominal flow where intervention is necessary for exception handling purposes. Including Robustness Variation as a subkind of Functional Detail Variation seems to have both advantages and disadvantages. If Robustness Variation families are simply special kinds of Functional Detail Variation families, then it may become possible to apply all of the results concerning how to reason about the family in order to assure that all of its members have certain desirable properties. On the other hand, by including this large set of kinds of variants in the scope of what could constitute a Functional Detail variation family, it seems to become harder to identify which kinds of reasoning can be applied, and how hard it might be to apply that reasoning. Creating a larger number of kinds of variation relations that define families with relatively more restrictive membership characterizations seems to open the possibility for more effective technologies for reasoning about those families. Therefore, formally and precisely defining the variation relations that define each of these kinds of families is important, and a key goal of new research.

2.3 Performance Variation

The members of a Performance Variation family all provide the same functionality, but differ from each other in the speed with which they execute. For example, in the OS domain, commercial operating systems nowadays are typically offered as families of (at least) two variants: a 32-bit variant and a 64-bit variant. Because of the additional addressing space, the 64-bit variants typically offer great performance gains for native 64-bit applications and for computation-intensive, memory-hungry tasks.

Another common example is a file search capability that may require a great deal of time to locate a requested file, or may be able to do so very quickly. Differences in performance may be noted only in some circumstances (e.g. only when large numbers of files are being managed), or under all (or most) circumstances.

Note that some performance variation can be achieved through functional detail variation, for example by providing different variants with different sorting and searching capabilities. Other variants might well differ from each other in far more substantive ways, as in the previous example of operating systems having both 32-bit and 64-bit variants, or whenever there may be a need for basing the structure of different variants upon different decisions about how to distribute needed modular capabilities across a range of platform components.

It is interesting to note that although performance variation may sometimes manifest as functional detail variation, in other cases it be quite different, and may require changes to the underlying architecture–i.e. the family of variants may no longer share a core asset architecture. This suggests that there is an interesting relationship between functional detail variation families and performance variation families, and they may sometimes intersect one another. To precisely understand and manage such relationships, it seems necessary that all variation relations under consideration are carefully and formally defined.

2.4 Service Variation

The members of a Service Variation family differ from each other in the service providers they utilize to provide different services. For example, different variants of a system may need to provide different quality of service (QoS) based on the requirements for variation in the service level agreements of different variants.

A lot of modern operating systems focus heavily on providing external services as a way to add additional value to their products. Some examples include services for synchronizing the files on multiple computers over a network, or backing up documents to a secure remote server, or different technologies for editing documents simultaneously online with a team of collaborators. Often, these services can be composed and choreographed together as part of a system designed and built using the service-oriented architecture (SOA) paradigm. Each service can then be switched in and out as the system evolves and needs to provide slightly different functionality or new services can be added in to provide additional capabilities.

Outside the OS domain, many web-based enterprises also offer variants with varying services for different market segments. Based on whether the website is being accessed by a commercial client or a home user, different variants may provide varying QoS or built-in guidance to the services that users interact with directly. Service variation may result in variants that provide differing functionality as well: for example, SOA-based systems for professional users may employ more sophisticated services to provide more extensive functionality for some tasks than the functionality provided by more rudimentary services for home users.

2.5 Interaction-Based Variation

An Interaction-Based Variation software family comprises variants that provide identical functionality but interact with users in different ways. For example, variants of the system may need to employ different interface technologies, or provide different interaction modes on different platforms.

In the OS domain, premium, high-end variants of operating systems will often provide capabilities for interacting with the OS in many different natural languages. Even though the presentation of a different language to a user does not affect the architecture, efficiency, or functionality of the system, it may result in a different experience for the user.

Another example of interaction-based variation is the necessity to develop different user interfaces (UIs) for the OS based on the platform on which it needs to run, such as a desktop computer, a laptop computer, a tablet, a cell phone, or another portable device. The need to run satisfactorily on different platforms may very well impose different performance requirements, but there is a more fundamental need for the system to interact with its users differently on different platforms. The need to meet interaction needs across a range of platforms (e.g. text input versus point-and-click) is met most naturally by a family of interaction-based variants.

It is interesting to note that here too (as was in the case of Performance Variation families) some of the variation might be covered by an appropriately broad definition of Functional Detail Variation. That is, some of the variants needed to achieve some interaction-based variation may be able to do so simply by elaborating on some functions differently. But it seems that not all needs for interaction variation are likely to be met in this way. More exotic platforms will potentially offer opportunities for interaction that are simply not possible in more conventional platforms. Thus, variants intended to support such platforms would presumably offer more or different functionality, and indeed might require different architectures than those used by variants that will run on other platforms. Finally, we note that the need for this kind of variation is relatively commonplace, especially in cases of systems that benefit from providing a client-specific UI for different platforms, such as web-enabled interfaces, downloaded clients, or iPhone applications.

2.6 Functional Invariance

Member variants of a Functional Invariance software family provide exactly the same functionality (including all of the details of all subfunctions), but this functionality is implemented in different ways. That is, the variants look identical when they are considered as black boxes, but their implementations may differ, and they may possibly be based on different underlying architectures.

As systems grow larger and more complex in the course of normal evolution, sometimes there is a need to reconsider the original architecture as it may no longer be suitable for the efficient implementation of projected new functionality. When this happens, it can leads to the creation of a new architecture that continues to support exactly the same functionality as was used to support an earlier version of the system, but is now better positioned for future evolution. This is most often achieved through approaches such as "refactoring," or restructuring already written code to conform to a new, changed architecture and design. Sometimes, refactoring may also be guided by a desire to simplify an overly complex implementation. In any of these cases, when refactoring occurs, the goal is to change the structure of the code without changing its functionality. Therefore, functional invariance might be considered as internal structural variation. In this case, the core asset among variants would be some specification of the functionality while the architecture of each variant may be variable.

Formally considering functional invariance as a special kind of variation has its advantages; because the program variants are expected to be functionally equivalent, this kind of variation suggests the possibility of amortization of analysis and testing costs (e.g. black box test case generation) for those situations where the same test suites should still be applicable and appropriate for all variants. Although functional invariance does not fall into the traditional definition of variation as it has been considered within the SPLE community in the past, we suggest that its potential to suggest approaches to amortization of some kinds of testing and analysis costs indicate that there may be benefits to consider it as another formal kind of variation

2.7 Goal Invariance

The variants in a Goal Invariance software family all share a common goal, but variants may achieve this goal differently from one another. As with functional invariance, goal invariance families are defined based on a consideration of the aspects of the variants that remain the same (in this case the fact that they all share a common goal), as opposed to the aspects that vary. The goal shared by the members of a Goal Invariance family may be specified as a collection of requirements that the system must satisfy, or a set of constraints that must be met, either with respect to the output or deliverable of the system, or some intangible results when some of the outputs may not be tangible.

In the OS domain, a good example of goal invariance is the ability of a device to connect to the internet through ethernet cable, modem, or wifi seamlessly to the user. In this case, the user may not care about the mode of connection, as long as the device successfully establishes a connection. Therefore, getting online may be considered the software family's goal, and the different variants of the system may utilize different hardware or software functionality or both to connect.

Another example is a software family whose goal is to complete a system backup successfully. In such a family, one variant may provide backing up functionality to a local drive, while another variant may support backing up to a network drive.

3. VARIATION IMPLEMENTATION AND MANAGEMENT

In this section, each of the several kinds of variation presented is revisited with a brief discussion of how it may be implemented and effected, and, where appropriate, parallels to existing methods and approaches.

3.1 Functional Detail Variation

Functional detail variation is perhaps the most familiar kind of variation discussed, and it is often exemplified by families that consist of the different software products that comprise a software product line (e.g. as defined by [7]). Such a software product line can be modeled or implemented by using feature-based approaches (e.g. as described in [5, 10, 16]) that use commonality and variability analysis to identify core assets. The different members of the software product line could then be built by combining different features in different ways, based on a common underlying architecture. Thus such a software product line may be considered an example of what we have described as a Functional Detail Variation family.

We note that a Functional Detail Variation family might also be implemented by the appropriate application of aspect-oriented programming at the code level. Specifying cross-cutting changes in functionality, or specifying the binding of different concrete implementations of an interface, are two such ways in which aspect-oriented programming approaches could be used to generate members of a functional detail variation family.

3.2 Robustness Variation

Because of the similarities outlined in earlier sections between functional detail variation and robustness variation, the latter could perhaps be thought of as a special case of the former, addressing the variation in the handling of exceptional situations (instead of the nominal flow), and typically driven by robustness requirements (instead of functional requirements) variation. Whether or not one kind of variation relation subsumes the other would depend on how both relations are formally defined. However, formulating suitable relation definitions requires careful consideration because these definitions may well determine how easy or difficult it is to reason about a functional detail variation family, a robustness variation family, or a family formed by the intersection of the two.

It is interesting to note that this relationship also means that the generation of a robustness variation family could be based upon the use of a core asset architecture that defines the nominal flow of all of the members of the family. Thus, one can imagine performing variability management in several steps, where a Functional Detail Variation family may be created first, then a separate Robustness Variation family might be created for different variants within that Functional Detail Variation family.

3.3 Performance Variation

Performance variation may under different circumstances lead to a family of variants sharing a common underlying architecture, a family of variants based on different architectures, or a family that comprises both a subset of variants that share an architecture and a subset of others that do not. Such complex variation relations clearly elucidate the need for formally understanding how all variants in a family are related. Furthermore, understanding the variation relation within a family may also facilitate understanding the relationship *between* software families. Families based on different kinds of variation relations may intersect, as in some of the suggested examples, such as a performance variation software family intersecting with a functional detail variation software family. Therefore, one variant may be a member of both families, and that variant may need to be subjected to different analyses depending on which family, and therefore which variation relation, is currently under consideration.

Such interactions between families based on different variation relations may have interesting implications for variation management and implementation. It may, for example, lead to a specification of precedence among different variation kinds such that assurances about one variation kind hold for other kinds as well; it may facilitate the navigation between multiple families that are interconnected through some common variants; or it may lead to easier generation of variants in a family, if this family is known to be related to some other families in well-defined ways.

3.4 Service Variation

In traditional systems, parallels could be drawn between components and services. That is, a component that is only responsible for providing some kind of utility can be thought of as a service provider, and then variations in this utility can be accommodated accordingly. From a technical standpoint, this may be achieved with aspect-oriented programming, where the service may be defined as an aspect whose behavior can easily be changed even though it is cross-cutting the entire system. However, such variation should likely be modeled differently from functional detail variation because SOA systems are often architected and designed sufficiently differently from traditional software systems and different requirements unique to the SOA domain may lead to different variation relations.

3.5 Interaction-Based Variation

Interaction-Based Variation families focus on modeling the relation that determines how different variants interact with users, even if this variation necessitates other kinds of variation. For example, to accommodate the processing of different modes of user interactions (e.g. command line input or mouse events), some functional modification of the software system may be required. However, this modification would presumably be contained mostly within the interface and controller components of the system, therefore being very different from typical functional detail variation.

Interaction-based variation is important because the way in which different tasks or different artifacts are presented can matter a great deal and can have a measurable influence on the behavior and satisfaction of users–especially in the case of human-intensive, highly distributed collaborative systems. This presentation aspect is often overlooked and is frequently not included in formal specifications of the requirements for variation. Moreover, the user interface is often considered to be an aspect of software that is not easily amenable to quantitative evaluation. Therefore, UIs are often evaluated qualitatively, using user studies, surveys, and focus groups–techniques that are beneficial for evaluating the usability of the system, but may only provide subjective results. A more careful modeling approach that takes into account the specific variation relations in Interaction-Based Variation families may help to ensure that the UI components satisfy the same requirements as the rest of the system and are amenable to the same kinds of quantitative analyses. Such quantitative analysis can then be combined with traditional qualitative metrics to make objective determinations on what variants are the most effective for what user groups.

3.6 Functional Invariance

Functional invariance is challenging because of the need to ensure semantic equivalence between variants, which is in general an undecidable problem. Although some progress has been made in trying to determine the absolute functional equivalence of some programs with formal analysis techniques, e.g. [14], in most cases the verification of equivalence falls on the developer and is achieved only approximately through repeated testing to ensure that the functional behavior of the software has remained unchanged.

By formally defining a set of properties that determine and constrain the behavior of the system so that two variants need only be equivalent with respect to these properties, it becomes much easier to check for equivalence. Note, however, that equivalence in this case is ascertained only with respect to these predefined properties, which the developer would have to specify. For example, the developer may formally specify certain functional requirements in a form suitable for static analysis, and then perform such analysis on many variants of the system. Even though these variants may be structurally quite different, they should all satisfy the same functional requirements.

3.7 Goal Invariance

Goal invariance is an interesting relation as it is not orthogonal to the other variation relations described in this paper. This presents a clear conceptual problem–variation among variants in the family may be more far-reaching and spread out than the other kinds of variation presented in this paper, and may affect all components of the system. In some cases, goal invariance seems to encompass several of the other kinds of variation discussed in this paper, such as functional detail variation, robustness variation, and functional invariance. If there is a demonstrable composition relation between goal invariance and some of the other kinds of variation, then goal invariance may provide a higher-level abstraction for reasoning about variation.

As noted, however, it is not only families defined using a goal invariance relation that seem to be interrelated with other kinds of variation relation software families. Understanding the meaning of these different family interrelations and devising effective strategies for variation management in such cases may facilitate analysis, since reasoning about goal invariance and other such possibly compositional variation relations may entail simultaneously reasoning about more than one kind of variation.

4. RELATED WORK

There has been extensive work on creating, using, and managing software product lines and software families [7, 15, 17]. Different approaches for variability management suggest handling variability at different stages in the software development lifecycle. For example, [4] propose explicit modeling and management of architecture variability, since the architecture stage is very ripe for addressing variation; this work also outlines different sources of variation, some of which correspond to some of the kinds of variation that we have observed and outlined here in this paper. Conventional modeling approaches, such as the use of UML to identify patterns in architectures [9], have also been applied to model variation in system architectures. There are clear benefits to such approaches for modeling variation and variability throughout the development life cycle. However, note that some of the kinds of variation presented in this paper seem to be inherently different from those addressed in previous literature. Moreover, some of the kinds of variation addressed here may be difficult to model using a single approach. In fact, they may necessitate the application of several different modeling and accommodation techniques. The approach presented in this paper is also being extended to comprise a way to formally define and classify the different kinds of software variation. One goal in doing this is to improve classification and communication about software variation and software families. Another goal is to be able to show that formal analysis and verification approaches may be effective in assuring that all members of a family must always necessarily have certain kinds of desirable properties.

Feature diagrams (e.g. [10, 18]) are widely used to model different feature configurations through variation points and different semantics for composing and combining features based on predefined constraints. Domain-specific feature graphs for specifying variability within a model, such as FORM [10] have conceptual goals that are similar to ours. Similarly, decision models (e.g. KobrA [3] and FAST [25]) can be used for instance generation and variant modeling, where variation points are indicated as decisions and, based on the selection, the model is extended with different sets of features. But in our work we suggest representing families implicitly, by means of formalisms such as Boolean functions that define a family without needing to enumerate all of its members. Feature graphs and decision models, on the other hand, are explicit enumerations of all possible mandatory and optional features. Such approaches can lead to very large feature graphs when all elaborations are included. And, indeed, some families may have an infinite number of members. The approach presented in this paper could likely benefit by being augmented with explicit enumeration approaches in some cases to enunciate how each product within a software family can be derived. Since the underlying goal of our work, however, is to better understand the different kinds of variation that exist in systems, we suggest the potential necessity to model these different kinds of variation both implicitly and explicitly, especially since our preliminary work suggests that specific techniques and methodologies may be more effective in defining and managing families characterized by different kinds of variation.

Other approaches focus on supporting variability modeling and management throughout different stages of the software development lifecycle. For example, the COVAMOF variability modeling framework [20] promotes the careful modeling of variation points and the different dependencies that may exist among variants. Hybrid approaches have successfully combined feature-oriented programming (FOP) and model-driven development to demonstrate how software product lines can be modeled (with a focus on features), and then products can be derived from these models using FOP [22]. The Koala framework [23, 24] has also been applied to support the reuse of components for different products within a family, and between different software families. Apel et al [1] have shown how model superimposition can be applied to different UML models: models are first decomposed into features, then appropriate model are composed together using superimposition to produce models for single variants within a product line. Such approaches provide important support for understanding how different variability management choices interact to produce different variants. Instead, our work focuses on understanding the underlying differences in the variation relations between all the variants in the family.

There are also several approaches for supporting variability management and software family realization at the implementation level, such as using component-based and generation approaches to specify initial configuration specifications of components and then to apply generation techniques for parameterization within the software domain [6, 8, 13]. There are also techniques and tools that have been successfully used for software family or product line implementation at the code level, such as feature-oriented programming [5], superimposition [2], aspect-oriented programming [12], mixin layers [21], or annotation and pruning approaches [11]. Although such techniques and tools are clearly useful and applicable during the implementation of different members of a software family, the code alone is only one of several components of a member of a software family. The approach presented here advocates careful modeling and reasoning based upon pre-code artifacts that seem particularly useful as they become available for exploitation earlier in the software development life cycle.

5. FUTURE WORK

As outlined, an approach for identifying different kinds of software variation could lead to better variability management and facilitate identification of appropriate constructs for the accommodation of variation. In order for a variation taxonomy such as the one suggested by the examples provided in this paper to be truly useful, however, the underlying relations that define family set membership need to be defined formally in a way that would support automatic construction of new architectures and canonical transformations from one variant to another. A conceptual framework is currently being developed and used to rigorously define the different kinds of software variation relations presented here.

Additionally, formal definitions of the variation relations that define different software families would encourage and perhaps facilitate reasoning about how different families that share variant members may interact, especially in the case of variation relations that are not orthogonal (such as composite functional elaboration variation and robustness variation families discussed in this paper). Such interfamilial interactions may also inform architectural decisions since some kinds of variation relations seem to accommodate the sharing of a common architecture, while others seem to result in variants that are structurally different and require different architectures.

The examples presented in this paper are from operating systems and other software system domains, but the authors originally observed these different kinds of variation relations in the context of process-guided application software. An approach for dealing with process variation was previously proposed in [19], and this work is a continuation of those efforts. In the course of studying variation in processes, it has become more apparent that there are few differences between process variation and general system variation, so the authors intend to continue exploring both domains to identify more variation relations and to validate the applicability of the formal relation definitions within the conceptual framework currently under development.

In addition to providing benefits to variability management and variant generation, such a conceptual framework could also be the basis for formal reasoning that could provide analytic results applicable to all the different variants in a software family. Future research is needed to understand better which kinds of analysis can provide which sorts of assurances about different kinds of software families. In particular, as noted earlier in this paper, a very broad definition of a kind of software family might offer the benefit of conceptual uniformity, but the disadvantage of making reasoning more difficult. Future research is needed to explore this tension.

6. CONCLUSION

This paper presents an approach for classifying different kinds of variation based on needs that arise from requirement for variation in order to meet diverse needs. The paper provides several examples of these kinds of variation, and then suggests the kinds of formal relations that might be used to define criteria for inclusion of variants in such software families. The classification of different kinds of needs for variation and different approaches to achieving it are important because it would allow software architects and developers to precisely model the variation and then reason about it. Doing so as early as the requirements specification, and carrying the formal modeling and classification of different variation relations through to the architecture specification and beyond could allow for better planning and could facilitate analysis of entire software families early on. In addition, early identification of mistakes in the modeling of variation could lead to timely remedial action.

Great strides have been made in the SPLE community to identify approaches that facilitate the generation and management of software families and product lines, leading to better utilization, increased effectiveness, and reduced costs. Carefully and formally defining the different variation relations that exist between variants in a software family may lead to even further gains in ease of variability management, variant generation, and reasoning about entire families of products as a whole, making assurances about safety, robustness, or performance requirements, among others.

7. ACKNOWLEDGEMENTS

The authors thank Stefan Christov, Lori Clarke, George Avrunin, and Alexander Wise of the University of Massachusetts Amherst for their insightful comments.

This research was supported by the U.S. National Science Foundation under Award Nos. IIS-0705772, CCF-0820198, and CCR-0427071. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the U.S. National Science Foundation or the U.S. Government.

8. REFERENCES

[1] APEL, S., JANDA, F., TRUJILLO, S., AND KÄSTNER, C. Model superimposition in software product lines, 2009.

[2] APEL, S., KASTNER, C., AND LENGAUER, C. FEATUREHOUSE: language-independent, automated software composition. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 221–231.

[3] ATKINSON, C., BAYER, J., AND MUTHIG, D. Component-based product line development: The KobrA approach. In *Proceedings of the the First International Software Product Line Conference (SPLC)* (Denver, CO, 2000), pp. 289–309.

[4] BACHMANN, F., AND BASS, L. Managing variability in software architectures. In *SSR '01: Proceedings of the 2001 Symposium on Software Reusability* (New York, NY, USA, 2001), ACM, pp. 126–132.

[5] BATORY, D. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 702–703.

[6] BATORY, D., AND O'MALLEY, S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology 1*, 4 (1992), 355–398.

[7] CLEMENTS, P., AND NORTHROP, L. *Software Product Lines–Practices and Patterns*. Addison-Wesley Professional, 2001.

[8] CZARNECKI, K., AND EISENECKER, U. Components and generative programming. In *ESEC '99/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (1999), pp. 2–19.

[9] GOMAA, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, Redwood City, CA, 2004.

[10] KANG, K. C., KIM, S., LEE, J., KIM, K., SHIN, E., AND HUH, M. FORM: A feature-oriented reuse method with domain specific reference architectures. *Annals of Software Engineering 5*, 1 (1998), 143–168.

[11] KÄSTNER, C., APEL, S., AND KUHLEMANN, M. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ACM, pp. 311–320.

[12] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming* (1997), vol. 1241, Springer-Verlag, pp. 220–242.

[13] KNAUBER, S. Synergy between component-based and generative approaches. In *ESEC '99/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (1999), pp. 2–19.

[14] PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. Differential symbolic execution. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2008), ACM, pp. 226–237.

[15] POHL, K., AND METZGER, A. Variability management in software product line engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software engineering* (New York, NY, USA, 2006), ACM, pp. 1049–1050.

[16] PREHOFER, C. Feature-oriented programming: a fresh look at objects. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming* (1997), vol. 1241, Springer-Verlag, pp. 419–443.

[17] SCHMID, K., AND VAN DER LINDEN, F. Introducing and optimizing software product lines using the FEF. In *SPLC '09: Proceedings of the 13th International Software Product Line Conference* (2009), Carnegie Mellon University, pp. 311–311.

[18] SCHOBBENS, P.-Y., HEYMANS, P., AND TRIGAUX, J.-C. Feature diagrams: A survey and a formal semantics. In *IEEE International Conference on Requirements Engineering* (2006), IEEE Computer Society, pp. 139–148.

[19] SIMIDCHIEVA, B. I., CLARKE, L. A., AND OSTERWEIL, L. J. Representing process variation with a process family. In *Software Process Dynamics and Agility: Proceedings of the International Conference on Software Process* (2007), Q. Wang, D. Pfahl, and D. M. Raffo, Eds., vol. 4470 of *LNCS*, Springer, pp. 109–120.

[20] SINNEMA, M., DEELSTRA, S., NIJHUIS, J., AND BOSCH, J. COVAMOF: a framework for modeling variability in software product families, 2004.

[21] SMARAGDAKIS, Y., AND BATORY, D. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology 11*, 2 (2002), 215–255.

[22] TRUJILLO, S., BATORY, D., AND DIAZ, O. Feature oriented model driven development: A case study for portlets. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 44–53.

[23] VAN OMMERING, R. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (Los Alamitos, CA, USA, 2002), IEEE Computer Society, pp. 255–265.

[24] VAN OMMERING, R., KRAMER, J., AND MAGEE, J. The koala component model for consumer electronics software. *IEEE Computer 33*, 3, 78–85.

[25] WEISS, D. M., AND LAI, C. T. R. *Software product-line engineering: a family-based software development process*. Addison-Wesley, 1999.