Characterizing Process Variation (NIER Track)

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

A process model, namely a formal definition of the coordination of agents performing activities using resources and artifacts, can aid understanding of the real-world process it models. Moreover, analysis of the model can suggest improvements to the real-world process. Complex real-world processes, however, exhibit considerable amounts of variation that can be difficult or impossible to represent with a single process model. Such processes can often be modeled better, within the restrictions of a given modeling notation, by a family of models. This paper presents an approach to the formal characterization of some of these process families. A variety of needs for process variation are identified, and suggestions are made about how to meet some of these needs using different approaches. Some mappings of different needs for variability to approaches for meeting them are presented as case studies.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.9 [**Software Engineering**]: Software Management—*variation*

General Terms

Design, Languages, Management

Keywords

process families, system variation, software product lines

1. INTRODUCTION

Software systems must often meet stringent requirements for variation across several dimensions. Thus, for example, software may need to run on different platforms, exhibit different ranges of functional behavior for different customers, meet different robustness requirements based on different deployment scenarios, and present capabilities to different users in different ways. When these requirements become very diverse and stringent, they are often met most successfully by producing a family of systems, rather than a single system into which elaborate kinds of flexibilities have been built. Such families of software systems are often referred to as software product lines, or software families [5, 15]. Our previous research has demonstrated that models of real-world processes have requirements for variation that are similar and indeed may be still more diverse and challenging [17]. In this paper we identify several examples of the kinds of variation that are needed in processes, and suggest approaches to meeting such needs through a variety of process family approaches. We suggest, further, that many of the kinds of needs for variation presented here are needed for other kinds of software systems as well, suggesting that these approaches should be of interest to system developers as well as process modelers.

In earlier work [18], we suggested that there are different canonical approaches that are useful in meeting different kinds of requirements for variation, and we suggested that any such approach should facilitate: *Generation*—the creation of new family members, *Navigation*—the identification of family members that best satisfy a set of given needs, and *Analysis*—reasoning about an entire family to prove that all family members have desired properties. A particular approach to creating a family might facilitate only some of these desiderata. Thus, for example, some approaches may make it easy to generate a new family member but may impede navigation or analysis. Conversely, a family might be defined in such a way that it is easy to reason about certain properties of the family as a whole thereby supporting the generation of future variants that are known to be "safe," but this may require onerous restrictions on the size and nature of the family. We believe that there is considerable value in identifying different approaches to generating families, and to understanding which types of variation requirements they meet and what advantages they offer. In this paper, we present a conceptual framework that considers variation in different dimensions derived directly from a requirement specification. Some mappings to different approaches for meeting such dimensions of variation are also presented, along with indications of the how well families generated by these approaches can be expected to meet the three goals of generation, navigation, and analysis.

2. PROPOSED APPROACH

Terminology. This paper strives to use the terminology used in software product line engineering (SPLE), with appropriate parallels to the process definition domain. Specifically the term *process family* is used to denote a collection of process definitions that are closely related based on a specific variation relation. Each member of a process family is called a *variant*. Variants may 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 subprocesses, agent behaviors, or other

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

artifacts), these commonalities are referred to as *core assets*. The activity of determining what variants best fit together in a process family and how to most effectively design the family in order to maximize reuse is known as *commonality and variability analysis*.

2.1 Classification

Below, different kinds of variation are presented and placed into a proposed classification. This list is not exhaustive, but is intended to suggest the range of variation observed in such diverse process modeling domains as elections, dispute resolution, and health care.

- *Functional Variation:* Variants differ in the details of one or more of the different functional capabilities specified.
- *Functional Invariance:* Variants provide exactly the same functionality (including identical behavior of all subfunctions), but this functionality is implemented in different ways.
- *Goal Invariance:* Variants all share a common goal, but they may achieve this goal differently. As with functional invariance, goal invariance families are defined based on an aspect that stays constant (in this case a common goal), as opposed to aspects that differ. Goals may be specified as a collection of requirements to be met, or constraints to be satisfied.
- *Robustness Variation:* Variants differ in the ways in which they are able to recover from incorrect or abusive use.
- *Performance Variation:* Variants all provide the same functionality, but differ in the speed with which they execute.
- *Agent*[1] *Variation:* Variants differ from each other in the agents they utilize to provide different functional capabilities.
- *Interaction-Based Variation:* Variants provide identical functionality but interact with users in different ways.

2.2 Conceptual Framework

To aid understanding, management, and implementation, variation can be viewed from two perspectives, namely a problem or requirements perspective and a solution or system perspective. This dichotomy is illustrated in Figure 1. From the problem perspective, variation is a collection of requirements for variation in performance, robustness, functionality, etc. These requirements may overlap, and may even be inconsistent. The way in which these requirements might be met is to be addressed from the system perspective. The system perspective focuses on studying how different approaches to achieving variation help to generate new variants, to support the navigation of a family, and to support analyses that apply to all variants in a family. A key contribution of our work is its suggestion of a problem-level variation meta model that can be used to connect different needs for requirements variation to appropriate solution-level family implementation approaches.

For example, consider an election process requirement for functional variation based on jurisdiction, where casting a ballot may entail different steps, involve different technology, and necessitate different levels of assistance from an election official. The variants in this functional variation family correspond to different elaborations of a ballot-casting subprocess that are invoked in specific higher-level election process execution contexts. Note that each elaboration of the ballot-casting subprocess can be thought of as a component since it defines a well-encapsulated set of procedures, it takes well-defined inputs, and produces well-defined outputs. Thus, the different elaborations correspond to different components that can be substituted in the overall election process, and steps or activities within a subprocess could be likened to method invocations. Such different elaborations of the ballot-casting subprocess entail differences in the nominal flow of the high-level election process definition. Hence in Figure 1 we connect the *Functional variation* dimension in the problem space to the *Nominal flow change* dimension in the solution space. The use of a solid arrow to make this connection denotes that this mapping from the problem level to the solution level facilitates two goals, in this case both the *generation* of new variants and also expedited *navigation*.

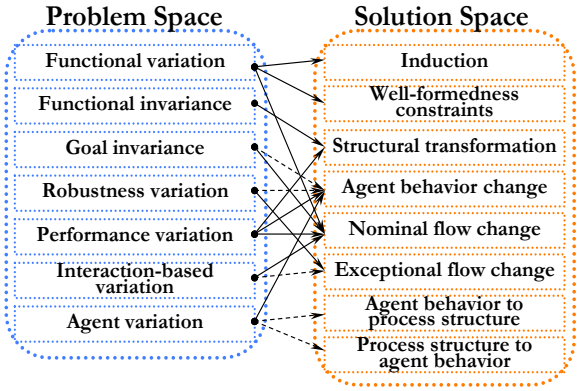

Figure 1: Example mappings from the abstract problem space to a solution space in a process definition language.

The dashed arrows in Figure 1 indicate mappings that may support only one of the three goals of navigation, generation, and analysis. For example, the dashed arrow from the problem-level dimension of *Goal invariance* to the solution-level approach of *Agent behavior change* indicates that in some cases it may be appropriate to use different agent behavior specifications to create process variants that achieve the same goal. Therefore, this mapping would help with the generation of new variants, but may not facilitate navigation as agent behavior specifications may be external to the process (this would also impede analysis). Due to space limitations, only the mappings from the *Functional variation* problem-level variation dimension are discussed in detail in this paper. Most of the remaining mappings, however, follow intuitively from the type of requirements the problem-level variation dimension covers and how those requirements are usually addressed; for example, as *Robustness variation* focuses on exception recovery, it naturally maps to *Exceptional flow change*, and other mappings follow similarly.

Another mapping that does not address all three goals is an inductive approach that facilitates solution-level family generation and navigation, but impedes analysis. To build families by induction, one can specify an initial common process definition core C, and a set of elaborators, E, and define a family Φ to be all process definitions that can be generated by a sequence of applications of elaborators from E to C initially, and inductively to any variant generated by applying a sequence of elaborators to C. E.g., let C be a single high-level process definition, and E be the replacement of any unelaborated step in C by any elaborating subprocess. Such a family may contain a variant in which a core function f_A, is replaced with a set of subprocesses, say f_{A1}, f_{A2}, and f_{A3}, whose combined capabilities achieve the required functionality of f_A (i.e. $f_A \rightarrow f_A(f_{A1}, f_{A2}, f_{A3})$). Although such a process family can

[1]A direct parallel can be drawn between agents in a process and service providers in a system built using Service-Oriented Architecture principles. Hence, we also use the term *Service Variation*.

easily be generated, and the generation procedure suggests how to facilitate navigation, it is not clear what analysis approaches could readily be used to ascribe properties to all members of this family. Depending on C and E, it may indeed be possible to construct variants that violate the well-formedness rules of the language in which C was written. This difficulty could be addressed by restricting family membership only to those generated processes that are well-formed. This approach complicates variant generation, but can facilitate analyses whose results are applicable to all variants.

Thus, consider using induction to generate process family members, but also using trace-equivalence to decide when generated variants are to be allowed to become family members. Specifically consider a functional variation process family Φ to be a collection of process variants p_i, determined to be trace-equivalent with respect to some event sequence property, T. Thus, for example all elaborations that can never cause an event included in T will qualify as members of the family. Then clearly all members of the family have the same attribute of adherence (or not) to T. I.e. $\forall p_i, p_j \in \Phi, p_i' \approxeq_T p_j'$ ($\approxeq_T$ denotes trace equivalence with respect to T). This is so because only events pertinent to the property are considered in determining trace equivalence, and thus two variants should reduce to trace-equivalent if and only if none of the elaborations in their variation contain events of interest with respect to the property, or the order of events coincides for both variants over all elaborations. More complex rules for rapidly determining family membership can be developed for cases where elaborations incorporate the possibility of certain kinds of event sequences that demonstrably cannot lead to differences in adherence to T.

3. RELATED WORK

There has been extensive work on creating and managing software product lines and software families [5,15]. Several approaches handle variation at what we refer to as the problem level, e.g. [14] proposes architecture templates for product lines and suggests that some templates facilitate creating new products (i.e. generation) or better analysis, but these goals are usually in conflict; [2] propose explicit modeling and management of architecture variability and outline different sources of variation with some similarities to the kinds of variation that we have observed in this paper, such as functional variation, performance variation, and agent/service variation. UML has been used to identify patterns in architectures and model variation in system architectures [7]. To facilitate quality assurance in the domain engineering phase of SPLE, which closely corresponds to the problem-level variation needs discussed in this paper, [8] focus mostly on what is referred to as functional variation in this paper, although they use UML activity diagrams augmented with formal Petri net semantics that allow for the careful and precise documentation of variability. A similar approach to address analysis at the domain engineering level is presented in [13], where the authors formalize domain artifacts as I/O-automata which are then model-checked against formally specified properties. This work focuses mostly on the problem level; our approach is similar, but contributes additional, and important, dimensions of variation, and strives to provide support for solution-level mappings to effect variability management and facilitate *generation*, *navigation*, and *analysis*. Additionally, some of the kinds of variation presented here, such as robustness variation and both invariance relations, seem to be inherently different from those addressed in previous literature, which mostly focuses on the use of features and on variation relations most closely related to functional variation.

Feature diagrams (e.g. [9,16]) model different feature configurations through variation points and different semantics for composing and combining features based on predefined constraints. Features closely correspond to the functional variation presented here. There are several approaches that focus on the problem-level specification of variation through features, such as using domain-specific feature graphs [9] or decision models [1]. Feature graphs and decision models tend to explicitly enumerate mandatory and optional features. The approach presented in this paper could likely benefit by being augmented with explicit enumeration approaches in some cases to enunciate how each product or process variant within a family can be derived. The underlying goal of our work, however, is to better understand the different kinds of variation that exist in systems and determine what approaches may be most effective in defining and managing families characterized by different kinds of variation. Hence, features may be insufficient for addressing variation driven by non-functional requirements, such as robustness, performance, or agent variation. Indeed, some of the kinds of variation presented here may necessitate the application of several different problem- and solution-level specification approaches.

At the solution level, component-based and generation approaches have been used to specify initial configuration specifications of components and then to apply generation techniques for parameterization within the software domain [6,12]; there are also techniques for product line implementation such as feature-oriented programming (FOP) [3], component reuse [20], aspect-oriented programming [11], or annotation and pruning approaches [10]. Although such techniques and tools are clearly useful for generating different members of a software family, the code alone is only one of several components of a member of a software family. Moreover, these approaches do not explicitly address process families and managing variation in processes. The approach presented here advocates careful modeling and reasoning based upon different process artifacts; some of these artifacts have clear parallels to artifacts produced in the software development lifecycle.

Other approaches focus on supporting variability modeling and management throughout different stages of the software development lifecycle through combining problem-level modeling of variation with solution-level product derivation. The COVAMOF variability modeling framework [19] promotes carefully modeling variation points and dependencies that may exist among variants. In [4], label transition systems are extended with features to describe the behavior of a family of systems and support model checking. These approaches are similar to the conceptual framework presented in this paper, and some of them map closely to both the problem-level functional variation (which most closely resembles features), as well as the solution-level nominal flow change. However, we would like to be able to reason about more than feature variation; considering features as the defining difference between variants may be necessary to address functional variation but may not be sufficient for the other dimensions.

4. FUTURE WORK

There are many research directions suggested by this work. Space limitations restrict us to discussing only a few. One obvious direction is to develop a more complete list of types of problem space variations and to devise an approach to classifying them. In addition, we are particularly interested in understanding the ways in which these different problem space variation families overlap and intersect with each other. We are also interested in identifying a range of approaches to generating solution space families. This paper has suggested a conceptual framework for generating such families, and indicated some possible specific approaches. Subsequent research will identify other approaches. For these different approaches we will investigate which of the three goals of *generation*, *navigation*, and *analysis* are facilitated.

Navigation facilitation is of particular interest because when intersecting families facilitate navigation, it should be relatively easy to traverse through multiple families via shared variants to reach variants satisfying specified combinations of requirements. For example, if a variant is a member of both a functional variation family and a performance variation family, then by specifying some functionality criteria, a developer should be able to navigate to this variant in the functional variation family, and then use it as a starting point to navigate to its performance variants. Such "chaining" of variants that may belong to more than one family could occur multiple times until a developer finds a suitable variant. Because the chain specifies a path that provides important information about not only the characteristics of the final variant, but also all the transformations that were applied to get to it, we call this operation *navigation* to distinguish it from a search operation that would not include any understanding of the provenance of the variant.

Analysis facilitation is another key direction of our research, aimed at identifying generation approaches that support specific kinds of reasoning about whole families of variants. In Section 2, we described how induction constrained by trace equivalence can be used to generate families whose members are all equivalent with respect to a pre-specified property. We also suggest that dynamic approaches such as simulation may be suitable for supporting analytic reasoning about performance variation families.

We plan to implement the tool suggested in Section 2. The proposed tool will support variation among processes defined using the Little-JIL process definition language. Little-JIL is a flexible process definition language that supports a considerable amount of variation within a single process definition, as it supports specification of complex coordination and concurrency, abstraction, resources, and agents, among other features [21]. But these features also seem useful as the basis for creating families that provide still broader amounts of variation. Providing initial support for creating families of Little-JIL process variants, which cover many commonly used process language features, should lead to useful insights about supporting variation in different process definition languages and other system representations.

5. ACKNOWLEDGEMENTS

The authors thank Lori Clarke, George Avrunin, and Alexander Wise of the University of Massachusetts Amherst, and Bashar Nuseibeh and Goetz Botterweck of Lero - the Irish Software Engineering Research Centre for their insightful comments. This research was supported by the U.S. National Science Foundation (NSF) 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. NSF or the U.S. Government.

6. REFERENCES

[1] ATKINSON, C., BAYER, J., AND MUTHIG, D. Component-based product line development: The KobrA approach. In *SPLC '00: Proc. of the 1st Int. Softw. Prod. Line Conf.*, pp. 289–309.

[2] BACHMANN, F., AND BASS, L. Managing variability in software architectures. In *SSR '01: Proc. of the 2001 Symp. on Softw. Reus.*, pp. 126–132.

[3] BATORY, D. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proc. of the 26th Int. Conf. on Softw. Eng.*, pp. 702–703.

[4] CLASSEN, A., ET AL. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE '10: Proc. of the 32nd Int. Conf. on Softw. Eng.* pp. 335–344.

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

[6] CZARNECKI, K., AND EISENECKER, U. Components and generative programming. In *ESEC '99/FSE-7: Proc. of the 7th Euro. Softw. Eng. Conf. held jointly with the 7th Int. Symp. on Found. of Softw. Eng.*, pp. 2–19.

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

[8] HEUER, A., ET AL. Formal definition of syntax and semantics for documenting variability in activity diagrams. In *SPLC '10: Proc. of the 14th Int. Conf. on Softw. Prod. Lines*, Springer-Verlag, pp. 62–76.

[9] KANG, K. C., ET AL. FORM: A feature-oriented reuse method with domain specific reference architectures. *Ann. of Softw. Eng. 5*, 1 (1998), 143–168.

[10] KÄSTNER, C., APEL, S., AND KUHLEMANN, M. Granularity in software product lines. In *ICSE '08: Proc. of the 30th Int. Conf. on Softw. Eng.*, pp. 311–320.

[11] KICZALES, G., ET AL. Aspect-oriented programming. In *ECOOP '97: Proc. of the 11th Euro. Conf. on Object-Oriented Program.*, vol. 1241, Springer-Verlag, pp. 220–242.

[12] KNAUBER, S. Synergy between component-based and generative approaches. In *ESEC '99/FSE-7: Proc. of the 7th Euro. Softw. Eng. Conf. held jointly with the 7th Int. Symp. on Found. of Softw. Eng.* (1999), pp. 2–19.

[13] LAUENROTH, K., POHL, K., AND TOEHNING, S. Model checking of domain artifacts in product line engineering. In *ASE '09: Proc. of the 24th Int. Conf. on Autom. Softw. Eng.* (2009), IEEE, pp. 269–280.

[14] PERRY, D. Generic architecture descriptions for product lines. In *Proc. of Dev. and Evol. of Softw. Arch. for Prod. Fam.* (1998), Springer, pp. 51–56.

[15] POHL, K., AND METZGER, A. Variability management in software product line engineering. In *ICSE '06: Proc. of the 28th Int. Conf. on Softw. Eng.*, pp. 1049–1050.

[16] SCHOBBENS, P.-Y., HEYMANS, P., AND TRIGAUX, J.-C. Feature diagrams: A survey and a formal semantics. In *RE '06: Proc. of the 14th Int. Conf. on Req. Eng.* , pp. 139–148.

[17] SIMIDCHIEVA, B. I., CLARKE, L. A., AND OSTERWEIL, L. J. Representing process variation with a process family. In *ICSP '07: Proc. of the Int. Conf. on Softw. Process*, vol. 4470 of *LNCS*, Springer, pp. 109–120.

[18] SIMIDCHIEVA, B. I., AND OSTERWEIL, L. J. Categorizing and modeling variation in families of systems: a position paper. In *ECSA '10: Proc. of the 4th Euro. Conf. on Softw. Arch.: Companion Vol.* pp. 316–323.

[19] SINNEMA, M., ET AL. COVAMOF: a framework for modeling variability in software product families. In *Softw. Prod. Lines* (2004), vol. 3154 of *LNCS*, Springer, pp. 25–27.

[20] VAN OMMERING, R., ET AL. The Koala component model for consumer electronics software. *IEEE Computer 33*, 3 (2000), 78–85.

[21] ZHU, L., ET AL. Desiderata for languages to be used in the definition of reference business processes. *Int. J. of Softw. and Informatics 1*, 1 (December 2007), 37–65.