Generation, Composition, and Verification of Families of Human-Intensive Systems

Borislava I. Simidchieva
bis@cs.umass.edu

Leon J. Osterweil
ljo@cs.umass.edu

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

ABSTRACT

Software products are rarely developed without providing different sets of features to better meet varying user needs, whether through tiered products as part of a product line or different subscription levels for software as a service (SaaS). Software product line approaches for generating and maintaining a family of different variants of software products address such needs for variation quite well. Real-world human-intensive systems (HISs) display similar needs for families of variants. A key contribution of this paper is to show how many of these needs can be rigorously and systematically addressed by adapting established techniques from system and software product line engineering (SPLE).

In this paper, we present an approach for creating such families by explicitly modeling variation in HISs. We focus on two kinds of variation we have previously described in other work—functional detail variation and service variation. We describe a prototype system that is able to meet the need for these kinds of variation within an existing modeling framework and present a case study of the application of our prototype system to generate a family in an HIS from the domain of elections. Our approach also demonstrates how to perform model-checking of this family to discover whether any variants in the family may violate specified system requirements.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software Management—*variation*; D.2.4 [**Software/Program Verification**]: Model Checking

Keywords

process families, system variation, software product lines

1. INTRODUCTION

The desiderata for software have changed tremendously as the industry has become pervasive and ubiquitous; most software released today is evaluated not only with respect to its functionality, but also with respect to its quality of service, usability, and other quantitative and qualitative metrics. The past approach of building a monolithic product to satisfy fixed and stringent requirements elicited from a limited number of users is no longer a viable business strategy. Today, different user bases expect different products that are customized to better solve their problems, run on their hardware, and accommodate their level of expertise. Operating systems only provide the features supported by the underlying hardware and enabled by the license the user purchased. Subscription and SaaS products provide tiered membership depending on the target demographic, and additional products are offered in several configurations from which buyers can make further choices.

Such differing needs are typically met by building *variants* of the software to meet different combinations of these needs. Increasingly complex and demanding requirements can be expected to cause the creation of ever larger sets of variants, but it is important, nevertheless, that these variants retain well-understood relations to each other. If such well-understood relations exist, the maintenance and further development of these variants and their encompassing *software family* or product line become much easier.

Complex, highly-distributed Human-Intensive Systems (HISs) exhibit similar needs for variation, but they also seem to require still further kinds of variation. HISs are similar to complex software in that they are large distributed systems, and like some software systems they can be safety-critical, but they present additional challenges in that they coordinate the actions of humans whose behaviors, nevertheless, reside within the system boundary. This adds complexity and suggests the need to adapt standard product line techniques in order for them to accommodate human variation. We refer to requirements specifications that mandate these various kinds of variation in HISs as problem-level variation. In previous work we have identified several kinds of problem-level variation [29] and suggested how each might be addressed and implemented by different kinds of solution-level variation techniques [27]. We continue to study this dichotomy, but in this paper, we focus only on solution-level variation. We demonstrate an approach to implementing two kinds of variation, and suggest requirements that these solution-level variation techniques seem useful in addressing. We demonstrate and evaluate these techniques through their application to an example HIS process drawn from the domain of elections. We use the term system process, or simply process, to refer to an HIS that coordinates the actions and activities of a set of human users and automated hardware and software components—under normal as well as exceptional situations—and the resources and artifacts they use. We make the case that the generation and analysis of HIS families, in which the users and their behaviors reside within the system boundary instead of outside of it (as is the case in most software systems), can benefit from the systematic application of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SPLC '14, September 15-19 2014, Florence, Italy
Copyright 2014 ACM 978-1-4503-2740-4/14/09...$15.00.
http://dx.doi.org/10.1145/2648511.2648533

approaches adapted from software engineering in general, and from SPLE in particular.

The two techniques for solution-level variation that we focus on in this paper are functional detail variation and service variation. Functional detail variation refers to variability within the implementation or elaboration of a certain part of a process in order to provide different functionality. This kind of variation is analogous to creating a family by providing different implementations of a given module within a system. Service variation refers to variability within the exhibited behavior of a single service. Within an HIS such as an election process, any agent can be considered to be a service provider. Thus, hardware or software components such as optical scanners or DRE (Direct-Recording Electronic) machines, and, more importantly, human agents performing their responsibilities as election officials, or voters themselves, all can be considered to be agents. This definition of service variation is an important distinguishing characteristic of our approach and allows us to consider variation in human behavior explicitly, yet within the boundaries of the system, which in turn allows for formal analysis. In our case study, we take the variation actually observed in real-world elections as a specification of required problem-level variation and show how we represent this variation at the solution level. We also show how the resulting solution-level process family can be formally analyzed to provide assurances about which variants in the family conform to which of various specified properties.

The rest of this paper is organized as follows. Section 2 describes functional detail variation and service variation as solution-level techniques, and indicates why they seem especially effective in addressing needs for problem-level variation in HISs. Section 3 revisits these two kinds of variation, and details how each may be best managed and implemented, presenting a real-world case study from the election domain. Section 4 shows how adaptations of software engineering analysis approaches can be applied to a process family presented in the previous section to identify variants that do and do not satisfy specified requirements. Section 5 discusses the applicability of the approach for generating process families and their subsequent analysis. Sections 6 and 7 contain a brief discussion of future directions and an overview of related work, respectively. Finally, Section 8 presents some conclusions.

2. APPROACH

Our approach supports the generation and analysis of solution-level process families that successfully address problem-level variation requirements. The approach entails creating a process family specification through elaborations at (potentially multiple) variation points in a common core process. We define a variation point to be a place where one or more different subprocesses or procedures can be invoked, specifying different ways this variation point can be elaborated. Once these variation points and their different elaborations have been identified, a process family can be created and leveraged to achieve a variety of goals; two we emphasize in this paper are 1) generation and 2) analysis. In the former, a process family specification can be used to generate a single process variant where each variation point has been resolved to exactly one elaboration, presumably based on some specific criteria. In the latter, a complete family definition is generated by elaborating all of the different subprocesses and procedures simultaneously while preserving well-formedness. Once generated this family can be navigated to facilitate selection of a specific variant, and analyzed to determine the extent to which all members of the family adhere to specified properties or constraints. All of this is greatly facilitated if, as in our work, this family specification is an actual executable definition of the system whose translation to an analyzable model is completely automated and hidden from the developer, eliminating the need for error-prone, time-consuming manual transcriptions of a system model into analyzable representations.

Meeting the need for such an executable language for specifying HIS families is challenging, but seems to be facilitated by languages with strong support for hierarchical elaboration. There are a number of process definition languages that support hierarchical decomposition and also provide facilities for specifying activity sequencing as well as artifact flow. In previous work, we have discussed how some problem-level variation requirements are nicely addressed by solution-level variation approaches that exploit hierarchy by building upon a common core [29]. We have also described some suggested mappings [27] that make it easier to identify what solution-level techniques can meet different problem-level needs. In this paper, we focus on solution-level support for functional detail variation and service variation, as they have been found to successfully address a multitude of needs for problem-level variation that arise in the election process domain, as well as in the domains of online dispute resolution and healthcare. We note that real-world processes exhibit diverse requirements for variation. Meeting these requirements is often best done by using solution-level variation approaches sharing a common core that incorporate a number of different variation points. These variation points may need to be hierarchically nested. If this is the case, variation points can be resolved hierarchically, starting with the initial variation points in the original common core and then resolving variation points that arise further down in the hierarchy. Thus, the original common core becomes augmented with higher-level family variants, and this newly-generated process family becomes the common core for the nested subfamilies. Strong support for specifying hierarchical decomposition clearly facilitates such composition.

To illustrate how solution-level definitions of functional detail variation can meet the needs for certain kinds of problem-level variation in real-world election processes, we focus on two different scenarios for voting—one using a paper ballot and another employing an electronic ballot supplied by a DRE machine. This functional detail variation is implemented using a common core with one variation point and two explicitly defined variants, one addressing the need for a voter to fill out and a cast a paper ballot and the other specifying how this interaction is done with touch screens. Such functional detail variation addresses functional concerns well, but other aspects, such as variation in how different agents perform activities, are not as well addressed.

Because humans are specified as agents *within* the system boundary in HISs, it is both useful and important to support specification of variation in human behavior (our work suggests this is no less important for non-human agents such as software subsystems). This has important implications for non-functional concerns such as privacy, security, and system interaction with the human. We illustrate how our approach supports the implementation of this kind of problem-level variation by constructing an example service variation process family comprising four variant behaviors of a DRE-machine in providing the "submit e-ballot" service.

Since functional detail variation points can result in variants that exhibit further functional detail or service variation[1] as noted above, we could generate a process family composed of several families nested within each other, causing a multiplicative effect. The combinatorial nature of these variants may quickly result in a large and unwieldy family (e.g., if two variants contain a functional detail

[1]Note: service variants specify agent behaviors for a single agent performing a task and therefore cannot contain variation points; functional detail variants can contain an arbitrary number of nested variation points and be composed with the resulting families.

variation point with four variants each, there are eight ways to select a single composed variant but 255 ways in which to compose a family containing one or more variant at each variation point, although some might be syntactically or semantically incorrect).

The multiplicative effect can also slow or impede analysis, but there are clear benefits to being able to reason about all variants at once (e.g. being able to assure that all variants conform to a specified constraint). Therefore, we allow for the explicit generation of an entire family in addition to just specific variants. Generating a specification for an entire family is somewhat similar to some annotation (tag-and-prune) approaches in SPLE where a conglomerate product can be built and the unnecessary features can then be pruned away [17]. Generating the entire process family allows a process developer to navigate the variation points to learn about different options, and allows the analyst to scan the entire family simultaneously, looking for variants that may have desirable (or undesirable) properties. We demonstrate an example of this kind of analysis in more detail in Section 4 of this paper.

By combining functional detail and service variation points, we can also explore multiple agent behaviors, both human and software, within multiple points in the process. Our case study illustrates service variation for a software agent but human agent behaviors can be similarly specified and used. Examining multiple service variants simultaneously, moreover, can also help identify malicious behaviors that can compromise the system.

3. CASE STUDY

In this section we present an example process family demonstrating both functional detail and service variation. We choose the Little-JIL process definition language [6,35] to demonstrate our approach because it meets the language requirements we outlined in the previous section, and other requirements (e.g. for strong support of exception management specification) whose importance became increasingly clear as we pursued the case study. Moreover, Little-JIL has been successfully used to specify several kinds of solution-level variations in a number of different domains. A Little-JIL process consists of three principal parts, namely specifications of: process activities, the artifacts that they use and produce, and the resources (including agents) that the activities require in order to be performed. A Little-JIL activity specification is a hierarchical structure of steps, with a parent step specifying the order in which its children are to execute. The steps create scopes that are used to support exception handling, with each step being able to specify how to handle exceptions of various types arising within the step's scope. A step may specify the types of artifacts it takes as inputs and produces as outputs, as well as the types of resources it requires in order to be performed.

Every step has one unique resource, called its agent, which is responsible for the performance of the step. Artifacts and resources are defined independently from the activity specification, providing excellent separation of concerns and allowing for different variation needs to be contained well. Steps, including their decomposition, and artifact and resource requirements, can be reinstantiated in different scopes providing a form of procedural abstraction. Little-JIL activity decompositions are usually depicted diagrammatically. The semantics of Little-JIL are defined rigorously by means of finite state machines, which can also be used to support the execution and formal verification of Little-JIL process definitions.

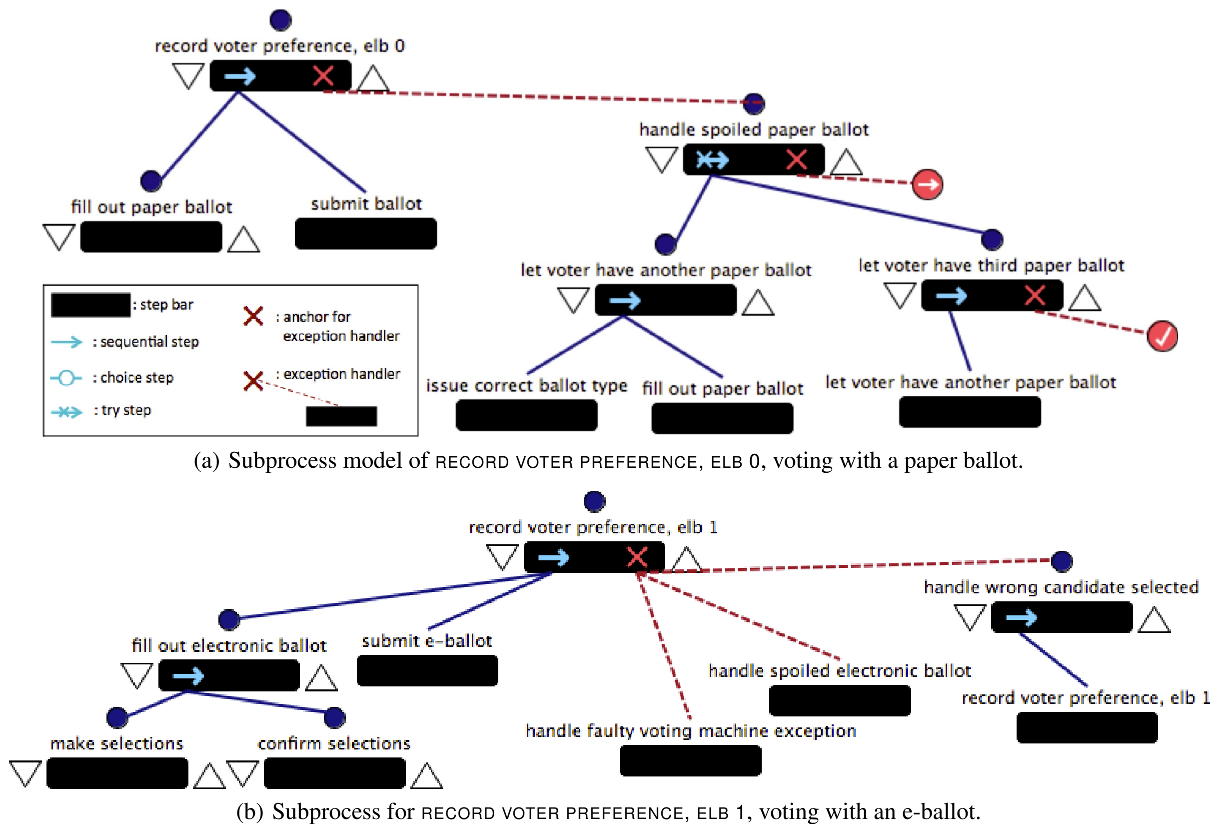
3.1 An election process in Little-JIL

Elections are a cornerstone of the democratic process in countries such as the United States. Every citizen of the US above the age of 18 is entitled to vote. Although all elections held in the US must satisfy many general requirements (e.g. no voter may vote more than once, only eligible voters may vote), there are additional requirements that may vary between different districts. For example, all voters must always identify themselves to an election official, but different districts handle voter identification very differently. Additionally, some districts employ DRE machines, while in other districts voting is done by marking paper ballots, which are then read either by election officials or by automated scanners. This suggests how US election processes can be partitioned into families. We now present a case study that shows how our approach supports the solution-level representation of a small election process family, using Little-JIL as a vehicle for this demonstration.

3.2 Functional detail variation

Figure 1 shows a process family addressing the functional detail variation need we discussed earlier, specifying that voting can be carried out with either paper or electronic ballots. The process diagram in Figure 1(a) specifies that the root step is to be executed by executing its children in sequential order, as indicated by the arrow badge in the parent step bar. Hence, a voter will first FILL OUT PAPER BALLOT, and then SUBMIT BALLOT. The agent and artifact specifications, and the details of what exceptions steps can throw, are not shown in the diagram to avoid visual clutter, but they are an integral part of a Little-JIL specification, as with any well-formed process defined in a language with rigorous semantics. For example, the step FILL OUT PAPER BALLOT produces as an output a PAPERBALLOT artifact, which is passed as input to the SUBMIT BALLOT step. We have observed the need to allow for variation in other aspects of a process apart from its activity structure, for example within the structure of, and access to, certain artifacts. For example, the same election process can specify problem-level variation needs for employing, within different scopes and contexts, provisional and regular, paper or electronic ballots, all of which may be accessible to different agents at different times. To maintain well-formedness at the solution level, a process family would have to accommodate this variability. In Little-JIL, artifacts are specified as JavaBeans, allowing the full flexibility of Java for their backend implementation. Thus, for example, all of the aforementioned types of ballots could be specified to extend a basic BALLOT.JAVA class.

We have previously described process families based on the need to achieve different levels of robustness [27,29], which we defined as variation in the extent to which different variants are able to recover from different kinds and different degrees of incorrect or abusive use. Although we do not discuss robustness variation in detail here, we note that robustness variation at the implementation level can often be effected by varying the exception management of a process, especially if that is possible to achieve without disturbing the nominal control flow of the process. Little-JIL has rich semantics supporting the specification of sophisticated exception handling scenarios. Thus, for example in Figure 1(a), the HANDLE SPOILED PAPER BALLOT step is an exception handler, being specified as such by its connection to its parent's exception anchor badge via a dashed edge. Little-JIL exceptions are typed as they are in most programming languages, and this handler catches exceptions of the type VOTERSPOILEDBALLOTEXCEPTION. Exception handlers can be arbitrarily complex and include references (or invocations) of steps and subprocesses specified elsewhere in the process, providing good support for variation. In this case the handler is a try step (as denoted by the crossed arrow in the left section of the step bar), which is defined to mean that its substeps will be *attempted* in sequential order from left to right until one executes successfully. In simple terms, this exception handling pattern implements the requirement that a voter who spoils a ballot must be provided with a



(a) Subprocess model of RECORD VOTER PREFERENCE, ELB 0, voting with a paper ballot.

(b) Subprocess for RECORD VOTER PREFERENCE, ELB 1, voting with an e-ballot.

Figure 1: Paper and e-ballot voting subprocesses for RECORD VOTER PREFERENCE.

new ballot, but no voter should be provided more than three ballots.

Note also that the parent step is titled RECORD VOTER PREFERENCE, ELB 0, indicating that this is one of several possible variant elaborations of RECORD VOTER PREFERENCE. In fact, Figure 1(b) presents a different variant elaboration, RECORD VOTER PREFERENCE, ELB 1. In this variant, the voter performs FILL OUT ELECTRONIC BALLOT (further decomposed into first MAKE SELECTIONS, followed by CONFIRM SELECTIONS). This time a different artifact, namely an EBALLOT, is passed out of FILL OUT ELECTRONIC BALLOT and into SUBMIT E-BALLOT. As before, the CONFIRM SELECTIONS step can throw a VOTERSPOILEDBALLOTEXCEPTION. The details of HANDLE SPOILED ELECTRONIC BALLOT are elided from Figure 1(b), but they are analogous to the paper ballot case. Figure 1(b) also specifies exception handlers for two more exception types, namely WRONGCANDIDATESELECTEDEXCEPTION, and FAULTYVOTINGMACHINEEXCEPTION, allowing for two more exceptional scenarios, and a larger, more elaborate process family, when a DRE machine is used.

The hierarchical structure of Little-JIL makes it quite easy to create variation points by specifying a step instance and a set of different functional detail elaborations for that step instance. We have extended the Little-JIL language and its supporting Visual-JIL environment to generate these family members as well as entire process family specifications. Similarly, service variation families are created quite straightforwardly by modifying the specification of the agent and/or resources that are specified within the declarations of a step. Indeed, Little-JIL agent and resource specifications list the characteristics and capabilities of the agent or resources needed by a step, rather than any single explicitly-specified agent or resource, and thus already incorporate specifications needed to create service variation families. The service variants themselves can be specified using any of a number of technologies, such as Little-JIL processes, Java code, and web service APIs.

Note that the two variants presented in Figure 1 specify a very small part of the complete election process, which begins with specification of voter qualifications, proceeds through voter registration, casting of ballots, and completes with vote tallying, auditing, and possible recounts. We elide the details of the higher-level process within which these subprocesses are invoked for space considerations. We now consider how and why the two variants just described can form a functional detail variation family. These two variants of the system provide different levels of functionality for RECORD VOTER PREFERENCE; moreover, they share the same common core, namely the top-level election process (not shown here) within which RECORD VOTER PREFERENCE is executed. This seems reasonable, as functional detail variation families are intended to respond to high-level functional variation requirement specifications. In this case, the system requirement specifies the need for elections in which either paper or electronic ballots, or both, are used to vote. A key advantage of this functional detail variation family is that the common core can be exploited as the basis for formal reasoning, which can lead to assurances about the entire family, as we will demonstrate in the following section.

We now show how one of the variants of the functional detail variation family presented in Figure 1 can be further augmented with its own service variation subfamily by focusing on one of the substeps of the second variant of RECORD VOTER PREFERENCE, namely the step SUBMIT E-BALLOT in Figure 1(b).

3.3 Service variation

We previously noted that service variation family members dif-

fer from each other in the service providers they utilize for 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. Services for synchronizing the files on multiple computers over a network, or backing up documents to a secure remote server can often be composed and choreographed 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 or new services can be added to provide additional capabilities, thereby building service variation families. Parallels can be drawn between components and services in traditional systems, as components in such systems are only responsible for providing some kind of service with variations in this service being accommodated accordingly.

Within the election domain, requirements to incorporate the use of different devices for the recording of votes creates a clear need for service variation process families. Specifically, recall that the SUBMIT E-BALLOT step invoked in Figure 1(b) is to be performed by an agent of type DRE MACHINE. The set of all process definitions that differ only in the specific agent that performs the step SUBMIT E-BALLOT (different behaviors of different DRE MACHINES) forms a service variation subfamily within the RECORD VOTER PREFERENCE, ELB 1 functional detail variation family. Figure 2 specifies four such different DRE machine behaviors for executing the SUBMIT E-BALLOT step. For the sake of clarity, in this paper we present the agent behaviors implemented as Little-JIL process fragments. This allows us to consider them as a special case of functional detail variation where every step within a variation elaboration must be performed by the same agent. Reducing service variation to functional detail variation significantly improves family-wide analyzability as demonstrated in the following section.

In Figure 2(a), the DRE machine will first COMMIT TO REPOSITORY the E-BALLOT artifact, then ISSUE UNIQUE ID for the E-BALLOT, and then PRINT RECEIPT, perhaps in order to comply with election VVPAT (voter-verifiable paper audit trail) requirements. Figures 2(b) and 2(c) are two more variants that only execute a subset of these steps, as shown. Figure 2(d), however, shows a variant that is not at all typical. It consists of the DRE Machine first executing FAKE BALLOT, then non-deterministically choosing (denoted by the slashed circle step badge) whether to commit to the repository the new FAKEEBALLOT created in the eponymous step, or the voter's own EBALLOT acquired from FILL OUT ELECTRONIC BALLOT in Figure 1(b) Here, we specifically add a variant that defines a *malicious* agent behavior. We model such behaviors to show how analysis of process families can enable detecting how and when malicious behaviors can cause problems. Even if no one behavior can be shown to be dangerous on its own, appropriate analysis techniques can deduce when agents, who appear to be "safe", might potentially collude to jeopardize process integrity. This kind of variation can also be useful when applied to software systems that use external service providers as well. For example, consider if OpenSSL were modeled as a service provider within an authentication system and proper analysis had been performed as our approach advocates. In the cases where the Heartbleed bug[2] manifested, those variants could have been detected as malicious behaviors. Of course continuous improvement through the removal of faults is achieved through iteratively analyzing a family, identifying and implementing improvements, re-analyzing to ensure there has indeed been improvement, then deploying and iterating. We next demonstrate a full cycle of analysis-driven process improvement.

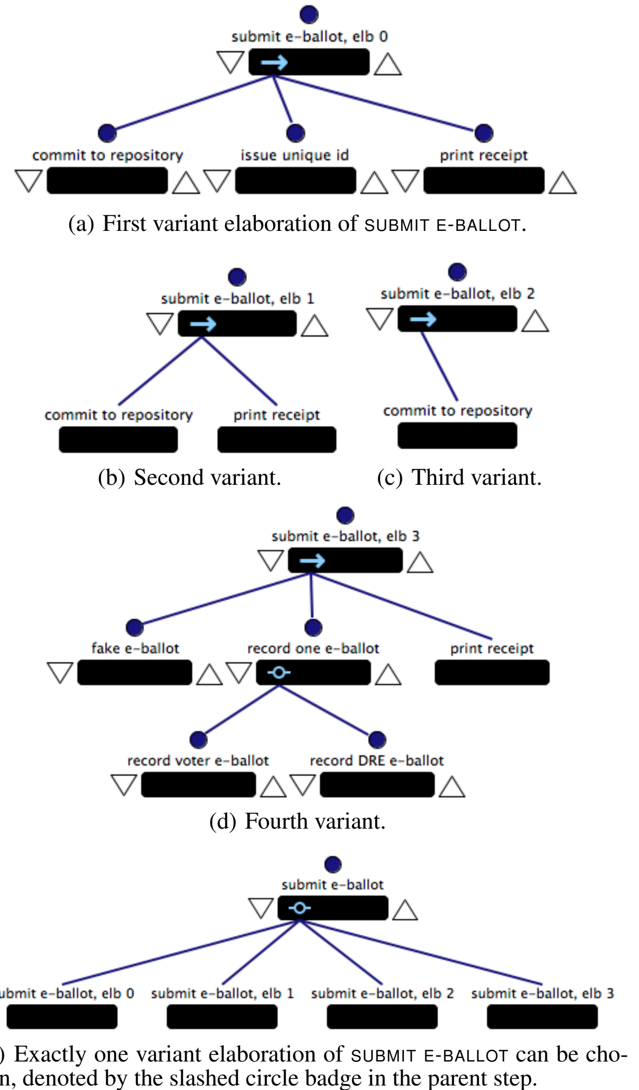


Figure 2: Service variation specifications for SUBMIT E-BALLOT containing different behavior description for DRE machines.

4. ANALYSIS

We indicated that our main goals in applying family-based approaches to HISs are to facilitate the generation of families, and to enable automated reasoning about some or all of the variants within a family. In this section, we discuss how we can apply a standard model checking approach to the nested election family we presented, consisting of two functional detail variation variants, and a service variation subfamily of four variants.

There are many requirements that a real-world election process, and therefore all variants within our process family, must satisfy. These requirements are usually specified at a high level of abstraction in an attempt to keep them independent from the specifics of how an election is carried out, partly because every jurisdiction in the US is free to define how it will carry out an election, as long as it does not violate federal requirements, such as the Help America Vote Act (HAVA). Some states such as California have fairly extensive election codes[3] including both procedures and requirements,

[2]http://heartbleed.com/

[3]http://leginfo.legislature.ca.gov

but many jurisdictions do not. These high-level requirements often do not cover specific technologies or protocols for recovering from exceptional situations. For example, consider the following requirements: 1) Only eligible voters can vote; 2) No one can vote more than once; and 3) The ballot the voter casts must match the voter's intent. These comprise a useful starting point for specifying election intent; however, to formally verify that all variants in a process family satisfy all of these (and perhaps other) requirements, the requirements must first be refined into lower-level, observable properties.

4.1 Requirement specification

We use PROPEL (PROPerty ELucidator [31]), a software tool that helps users formalize all the details associated with a certain high-level requirement, to precisely define the property that a ballot should correctly record the voter's intent. Voter "intent" is a very controversial term among election researchers; therefore for this paper we define "intent" simply to mean that if the voter made a selection on the ballot and then cast it, then the selection actually made by the voter is the voterÕs "intent". It is vital that no changes can then be made to that ballot before it is counted. We represent this as a property specified as the Finite State Automaton (FSA) shown in Figure 3, that defines the acceptable order in which events can occur for any trace through the process model.

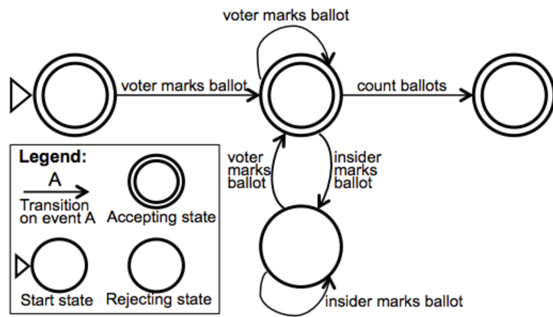


Figure 3: FSA for the property "After a *voter marks ballot* **event, no** *insider marks ballot* **event can occur until** *count ballots* **occurs."**

Each of the nodes in this figure represents a state in the FSA. Double circles indicate accepting states, and a single circle indicates a non-accepting state. Arrows between states are state transitions, and each is annotated with a corresponding "event" that triggers it. When a transition is triggered, the current state is updated to the target state of that transition. There are many different events that may occur in an election, but each property is typically only concerned with a small subset of them (the property alphabet). In this example the property alphabet consists of **voter marks ballot**, corresponding to a voter marking a ballot, **insider marks ballot**, indicating that a mark was made by an insider to the election process (someone other than the voter, such as a rogue election official or a compromised voting or counting machine), and **count ballots**, corresponding to the counting of votes. An FSA used for verification must be deterministic and total, meaning that for each state every event in the property alphabet must occur on exactly one transition leaving the state. If an event occurs that causes the sequence of events to be unacceptable, then the current state is updated to a non-accepting state, called the violation state from which all transitions are self-loops, causing the FSA to remain in that state until the process terminates. To keep the FSA representation in Figure 3 uncluttered, the violation state and all the transitions to it are not shown.

In Figure 3, the leftmost state is the starting state, denoted with an arrowhead. If the event **voter marks ballot** occurs, the current state is updated to the middle state on the top. From there, the voter may remark the ballot (HAVA requires that the voter be given at least one replacement if a ballot is spoiled, California mandates exactly three tries) as indicated by the self-loop marked with the **voter marks ballot** event, or the **count ballots** event might occur, moving to the right-most state and indicating that the property has now been satisfied. On the other hand, if the current state is the middle state on the top and the event **insider marks ballot** occurs, the current state is updated to the bottom, non-accepting state. From there, if **count ballots** occurs the property is violated (because the transition, not shown here, leads to the violation state), or, if **voter marks ballot** occurs again, we return to the state above (indicating that the voter might have the chance to fix the discrepancy before the ballot is cast). This FSA therefore specifies the requirement that the event **insider marks ballot** should never be allowed to occur between any pair of **voter marks ballot** and **count ballots** events.

For the system we have been discussing, a sequence of events that would drive this automaton to the violation state would indicate a malicious agent behavior on one or more variants of the service variation subfamily compromising the system. Because we can generate entire families simultaneously, and we can nest service variation families within functional detail families, we can consider the multitude of combinations of agent behaviors simultaneously, using analysis techniques such as model checking. Although in this example it is obvious that the malicious actions of one of the DRE machines modeled in Figure 2 can cause of violation of the property, it is not hard to conceive of more complicated processes in which different kinds of double-checking might be implemented to attempt to identify and defeat such behaviors. Dynamic testing approaches, usually black-box for election systems, would be complicated if, for example, the machine only occasionally replaces the voter's ballot, Thus, we advocate using model checking to formally verify that all variants of such a system satisfy a stated property. To do so, we must determine the correspondence between the events comprising the property's alphabet in Figure 3 and specific steps in specific variants within the election process family. This correspondence is explicitly defined through a binding between process steps and property events.

4.2 Model checking

Table 1 lists the bindings between the property alphabet events and steps in the process family. The event **voter marks ballot** is bound to two process steps, FILL OUT PAPER BALLOT and FILL OUT ELECTRONIC BALLOT. This means that when either of these steps completes successfully (depending on which variant is chosen), the event **voter marks ballot** occurs. Similarly the event **insider marks ballot** is bound to the step FAKE E-BALLOT and **count ballots** is bound to the SCAN VOTES step that occurs much later during the counting phase of the election process and is not shown in this paper. We now describe our model checking approach, in which automata representing process properties are driven through states as traversals of a process model encounter steps whose execution generates events in the automaton alphabet.

Because we have a formally specified property, formally defined process family, and corresponding bindings, we can apply formal model checking approaches such as Finite-State Verification (FSV) to determine if all variants in the process family satisfy the property. FSV first constructs a finite model reflecting all the sequences of the events in the property alphabet that could occur for all the possible

Property Event	Step Name
voter marks ballot	FILL OUT PAPER BALLOT
voter marks ballot	FILL OUT ELECTRONIC BALLOT
insider marks ballot	FAKE E-BALLOT
count ballots	SCAN VOTES

Table 1: Bindings between steps in the Little-JIL process family definition and events in the property alphabet.

traces through all of the variants in the process family (which in this case consists of all of the variants in the two nested process families presented in the previous section). FSV then checks if all the traces through all of the members of this family are consistent with the property specification [10, 15, 21]. We use the FLAVERS FSV engine [12], which was originally developed to verify Ada and then Java programs, but has been extended to include support for verifying Little-JIL process definitions. If the model satisfies the property (i.e., the FSA is in an accepting state when the verification completes), then all possible traces are consistent with the property. Since the set of all possible traces includes each possible trace through each process variant, we have thus proven that each variant satisfies the property. If the model violates the property (i.e. the FSA is left in a non-accepting state), the violation could indicate error or imprecision in the model, an incorrectly defined property, or it could indicate that some execution of one or more of the process variants indeed violates the property. FLAVERS produces a counter example trace through the model that demonstrates how the FSA can enter its violation state, and careful inspection of the FLAVERS counter example trace can help to identify the cause of the violation. Once the source is identified, appropriate corrections can be made and verification repeated until the property is satisfied by all variants. Although this paper focuses on one property, a given process family is usually expected to adhere to large suites of properties representing a variety of high-level requirements, and a tool such as FLAVERS is used to verify them all. Typically, if the error detected is a manifestation of an error in the real-world process, then the process definition is modified in consultation with domain experts. However, since we generate whole families of related process variants, we can also use FLAVERS as an exploratory tool to identify which variants are safe to use (at least with respect to the properties being verified), and which variants might cause problems and under what circumstances.

4.3 Results

Running FLAVERS on our election process family with the property from Figure 3 and bindings from Table 1 resulted in a violation. As noted before, when a property is violated, FLAVERS generates a counter example trace. In this case, the counter example is a trace through the process that contains the voter completing the FILL OUT ELECTRONIC BALLOT step (note, this is a substep of the RECORD VOTER PREFERENCE, ELB 1 step) and triggering the **voter marks ballot** event, then the DRE machine performing the FAKE E-BALLOT step, triggering **insider marks ballot**, and then the SCAN VOTES step being performed leading to the **count votes** event, which puts the property automaton in the violation state.

Having identified one specific variant that led to a violation, we then removed that variant from the service variation subfamily, generated a new subfamily only containing the behaviors from Figures 2(a), 2(b), and 2(c), and reran the verification. When we did that, the property indeed held for all possible traces through the family, indicating that all variants specified are safe with respect to this property. These results may seem straightforward, but consider what might happen when there are many more variants to consider, with only some of them being malicious. For example, looking back at Figure 1(a), note that we did not specify how SUBMIT BALLOT is to be executed. If we define a second service variation family specifying that SUBMIT BALLOT gets executed by an election official, we can then specify some honest and some dishonest behaviors for this human agent (for example where the election official replaces the voter's ballot with a pre-marked ballot). We leave out the details here for space considerations, but when we defined malicious behaviors for both SUBMIT E-BALLOT and SUBMIT BALLOT (and specified the additional bindings for the new variants analogously), FSV provided a new counter example trace. Having identified the violation and trace, election officials can then take steps (such as inserting double-checking by a second election official) to prevent the election official's dishonest behavior from causing a fraudulent ballot to be cast. Other offending variants can likewise be detected and dealt with until the entire family satisfies the property. We are then left with an election process family that is guaranteed to be safe for the property shown in Figure 3, which in turn guarantees that any process variant generated from that family is safe by construction.

5. DISCUSSION

So far we have presented a limited application of our approach to a composed process family comprising functional detail and service variation, and we have demonstrated how such families can be analyzed using a single analysis technique, finite-state verification. In this section we discuss and evaluate the suitability of the approach in supporting these as well as other endeavors.

5.1 Implementation

We outlined two distinct goals for generation, depending on the desired outcome—whether we want to generate a single variant or the whole family. The first goal was to support the instant generation of a specific process variant when the desiderata were clear and the process developer could make reasonable choices at all elaboration points. The second goal was to support the generation of an entire family of variants in order to facilitate navigation (the process developer stepping through the process model and being able to view different variants at different variation points in order to make a selection). Importantly, the latter generation goal also implied the ability to subject the whole family to a battery of analyses to determine which variants best meet various system requirements. Since we defined functional detail variants to share a common core and vary in the ways a certain step is elaborated, a hierarchical process specification, such as the one Little-JIL provides, seemed particularly useful in supporting the generation of such variants.

To support these generation scenarios, we extended the Little-JIL language to make variation a first-class construct, and then developed the Little-JIL Elaborator system, which provides additional capabilities to the Visual-JIL editor for dealing with variation. The elaboration step, a new step kind equivalent to a variation point, is a special case of a reference step in Little-JIL. A reference step is a placeholder indicating an invocation of a subprocess defined elsewhere. Reference steps must "resolve" to a single specification of such a subprocess to maintain well-formedness. When a single variant is generated, for every elaboration step in Little-JIL we only need to attach one specification of a functional detail variant. Since this is equivalent to a normal reference, after confirming that an elaboration step resolves to a single specification, the Little-JIL Elaborator handles it by resolving to the single elaboration. The second scenario, when an entire family is generated, must be handled more carefully and contains the main extension of the semantics of the Little-JIL language to allow for variation. In that case, multiple elaborations share the name of the elaboration step,

which would result in a process that is not well-formed (and therefore not suitable for analysis) without some additional transformations. In this case, the Little-JIL Elaborator automatically generates a choice step whose name is the same as the original elaboration step, renames all elaborations by appending a ", ELB N" (where N is an integer assigned in sequence as the elaborations are being processed) to the original step name, makes all elaborations children of the newly created choice step, and copies the union of the interface declarations to the new parent (to ensure artifacts are still being passed as expected). The elaboration step then "resolves" to the newly generated choice step which has all elaborations as its children. An example of this newly generated step is shown in Figure 2(e); an analogous parent was created for the two variants in Figure 1, but it is not shown here for space considerations.

The success of the Little-JIL Elaborator in meeting complex variation implementation needs was greatly facilitated by the orthogonal specification in Little-JIL of artifacts and agents with respect to activities. This provides excellent separation of concerns that facilitates variation specification along either orthogonal dimension, which proved to be helpful in addressing different problem-level variation needs. In this paper, we reduced service variation to a special case of functional detail variation by demonstrating agent behavior variants as Little-JIL suprocesses for clarity and improved analyzability. But, as noted, agents in Little-JIL can be human or automated, and agent behaviors can be specified in a variety of ways (e.g. as code snippets in Java). Little-JIL steps specify agents and resources as abstract requests of needed capabilities, and specific agents and resources are selected at runtime. This approach to returning sets of agents or resources in response to variation requests from steps can create service variation families. Family-level analysis is complicated when some members are implemented in different languages, but this difficulty can be addressed if the behaviors of these family members have been pre-analyzed and annotated. Artifact structures could also be varied to a reasonable extent by using Java's polymorphism capabilities, as was illustrated earlier with two subtypes of the BALLOT.JAVA declaration.

5.2 Experience

Once a variant or process family is generated using the Little-JIL Elaborator, various analyses can be performed on it. Similarly to generation, if a single variant is created then analysis is no different than it is for a normal Little-JIL process. When we are analyzing a family of related processes, analysis becomes more expensive, but some of the costs are also amortized by commonalities among family members. The properties are defined as usual because they are representation-independent, and can now be applied to even more process definitions because sometimes entire families need to satisfy the same requirements (as in the case of the aforementioned Yolo and Marin County election processes—both must meet all the California requirements in addition to the HAVA requirements).

Our experience has shown that an analysis engine can be used as an aid for variant exploration. If we represent agent behavior variation, we can analyze the variants to explore different ways to compromise the system, including insider attacks, agent collusion, or simply incompetent or out-of-sequence performance of critical steps. Seemingly small changes to behaviors can have large or long-lasting ramifications, as demonstrated by the property in Figure 3, where the same ballot is carried through the whole process until it is time for the ballot to be counted. It is not far-fetched to imagine how a DRE could be programmed to "malfunction" only part of the time; when tested as a black box, this behavior may well go unnoticed. The prior act of having a human bind steps to property events can make such problems clear without the need to carry out FSV. But the binding activity could be automated to add bindings for steps that modify (as opposed to just read) certain artifacts, or steps that request certain agents, in which case the problems would then be surfaced by FSV, as the output of a completely automated analysis.

Since process improvements can be quickly identified and deployed, a new process family can be regenerated and reanalyzed to see if it conforms better. The process developer can also prune away violating variants one by one until the property holds, identifying a safe subset of variants. This additionally encourages reuse because a collection of preverified elaborations could be safely composed if they meet certain requirements. Once a process developer has a selection of safe variants, this enables the selection of variants based on other attributes, facilitating the generation of process families based on robustness or performance variation, for example.

6. FUTURE WORK

There are many promising directions for future work, both within the generation and the analysis of process families. In terms of generation, since Little-JIL is a fully-executable language, the interpreter could be modified to support executing process lines to potentially support real-time deployments of self-adapting families. This would be especially helpful for process guidance systems, which could then self-optimize depending on the context or self-heal if exceptional situations arise that require the switch to a different robustness variant within the family. Additionally, our approach allows for the specifications of reusable process modules and components; the domain of elections is rich in process modules that are shared among jurisdictions. A library of standard procedures could support the generation of process variants through composing pre-specified (and pre-verified) modules for voter registration, ballot counting, and so on. Thus far, we have considered the election processes in Yolo and Marin counties in California, which are very different, but employ similar procedures for certain subprocesses.

Process families can be leveraged for other kinds of analysis besides FSV. Extending previous work [28], we have successfully applied fault tree analysis (FTA) to very large families of election processes; analyzing the resulting fault trees to aid in detecting fraudulent or colluding agents, as well as specific variants within which the misperformances of steps can have disastrous consequences. Using FSV and FTA as complementary approaches and finding a way to combine the analysis results would be especially helpful. Finally, process families can be used to evaluate different agent behaviors along multiple different aspects, such as performance metrics or expertise assignments, by running simulations. Additionally, simulations with random agent assignments can help to derive the impact of certain steps being executed by a rogue agent, or the likelihood that two steps can be executed by the same agent or multiple colluding agents for attack prevention.

7. RELATED WORK

The generation and management of software product lines have been extensively studied [9, 25]. For brevity, we only discuss approaches that provide support for variation at the solution level. At the design stage, some approaches support the modeling and configuration of variants based on composing features. Feature diagrams (e.g. [26]) employ variation points where different predefined constraints can be applied to combine features. Similarly, decision models such as KobrA [4] and FAST [34] support choosing features at decision points to generate variants. Our approach for nesting process families at the solution-level is similar to fea-

ture diagrams and decision models, however it results in a process family instead of a feature specification. At the implementation level, approaches for supporting variability management and product line generation tend to focus on composition and parameterization of components following a configuration specification, such as demonstrated in [11, 13, 20]. There are also techniques supporting code transformation, such as implementing product lines with feature-oriented programming and related modeling approaches [5, 18], aspect-oriented programming [1, 13, 19], or annotation and pruning approaches [17]. Although such techniques facilitate the generation of the source code implementing specific variants within a software family, the code is only one facet of a software product. Our approach supports the specification of a system architecture as a hierarchical decomposition into increasingly detailed modules and components. Maintaining this kind of pre-code artifact throughout system evolution can facilitate formal reasoning and the evolution itself.

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 [30] promotes carefully modeling variation points and dependencies that may exist among variants, addressing different aspects of generation, navigation, and analysis. The pure::variants tool[4] similarly provides support for the generation and navigation of new variants by supporting a configuration specification including different constraints at the domain engineering level (similar to problem-level variation not discussed here), and consequently derived configuration specifications for the implementation of variants at the application engineering level (similar to solution-level variation). Our approach builds upon our earlier work, which similarly aims to support the specification of variation at the problem-level, and provide clear guidance on how best to achieve that variation at the solution level. Little-JIL seems to be an especially appropriate vehicle for studying variation from requirements through analysis because of its rigorous semantics that support its use as an architecture specification language, but also allow for it to be executed and formally analyzed. Some of these previous approaches map closely to both the problem-level functional detail variation not discussed in detail in this paper (which most closely resembles features), as well as the solution-level process fragment elaborations illustrated in the case study presented. Considering features as the defining difference between variants may be necessary to address functional detail variation but may not be sufficient for the other dimensions we would like to support; service variation in particular as we define it to encompass human behavior variation is not well addressed in traditional software product line approaches.

Within system processes, there are also several approaches that support generating variants. Some approaches extend existing notations such as SPEM; [33] proposes an extension to support the specification of a process line architecture, and later derivation of project-specific process variants. The SPRINTT approach [22], provides a comprehensive framework for specifying and generating a process core and variants in vSPEM. Similarly, [2] presents an approach for software process line generation through scoping, by first analyzing existing organizational processes for commonalities and then organizing them within a decision model to allow for future instantiations of customized variants. As a recent literature review [23] notes, most existing approaches have serious limitations because they either only address one aspect of variation (most often what we describe here as functional detail variation), or, when including data-flow and role-based variation these are usually not varied orthogonally to the activity specification. We provide a framework that can address several of these aspects orthogonally, allowing for the composition and nesting of families based on different variation relations.

Additionally, we support formal analysis and verification. In most process line and process family approaches, the term analysis refers to commonality or context analysis, and verification often means little more than checking syntactic compliance and well-formedness to assure that all members of a family are "pre-verified" to be syntactically correct or otherwise well-formed. Our work has greater ambitions aiming to show that all family members are "pre-verified" to adhere to properties encoding desired behavior with respect to safety or other requirements, defined, for example, by FSAs. Others in software product line community share these ambitions. In this other work, a product line specification is typically manually translated into some formal notation amenable to analysis, and the model is then checked against a specified set of constraints. Our work differs in that our process families are immediately analyzable, being specified in a well-defined, executable language, thus eliminating the need to first manually translate them into another representation, which is time consuming and error prone. Some techniques focus on placing restrictions on the creation of new variants, thus impeding generation, but providing well-formedness assurances for all variants that can be generated (e.g., [16, 30, 32]). Others focus on the traceability of features to subsets of components from the core assets and can reason about those relationships using QSAT (a SAT solver modified to handle quantified Boolean formulae) [24], or, for more sophisticated analysis capabilities the system can first be modeled in product line CCS and then checked against multi-valued modal Kripke structures to determine legal configurations that satisfy the requirements [14]. Although the goals of such approaches are consistent with the goals of our analyses, their analyses usually address only well-formedness constraints, and not other types of properties, such as safety, performance or robustness properties. Service variation is addressed in [3] where a family is first represented using a modal transition system and then verified with respect to requirements specified in vaCTL. Asirelli et al. define service variation to be different web service providers (in this case, websites for flight and hotel bookings), whereas service variation in our approach encompasses those kinds of variation but also allows for variation in human behavior modeled within the system boundary of an HIS. In [8], transition systems are extended with features to describe the behavior of a family of systems and support model checking, thus providing important support for analysis, though generation and navigation are not a central focus of this work. This work is extended in [7] where further functional specifications of the system are written in the Text-based Variability Language (TVL) and fPromela (a featured extension of Promela) and then model-checked against fLTL properties to determine what products satisfy the properties.

8. CONCLUSION

In this paper, we present two specific kinds of problem-level variation we have observed in the real world and have subsequently modeled and analyzed at the solution level using the Little-JIL process definition language and the FLAVERS model checking engine. We discuss the benefits of considering variation within real-world HISs and demonstrate how we can adapt and extend existing approaches for managing software and system product lines to address variation needs within a new domain. We then focus on ser-

[4]http://www.pure-systems.com/

vice variation—variations of the behavior of humans or automated agents—which is largely ignored in software systems. We show how we can bring these agents within the system boundary by presenting a case study based on a real-world election process and give the reader intuition about how we can use our approach to make assurances about what agent behaviors are safe and what others may compromise the integrity of the system. Although we focus on human-intensive processes, we draw clear parallels addressing how this approach can be extended to apply to software and system product lines in the future.

9. ACKNOWLEDGEMENTS

The authors thank the Yolo County Recorder, Freddie Oakley, and her Chief Deputy, Tom Stanionis; and the Marin County Registrar of Voters, Elaine Ginnold, for their expertise during the elicitation of election process variation needs. The authors also thank Matt Bishop of UC Davis and Lori Clarke, George Avrunin, and Heather Conboy of UMass Amherst. This research was supported by the U.S. National Science Foundation (NSF) under Award Nos. IIS-1239334 and CNS-1258588 and the National Institute of Standards and Technology (NIST) under grant 60NANB13D165.

10. REFERENCES

[1] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[2] O. Armbrust, M. Katahira, Y. Miyamoto, J. Munch, H. Nakao, and A. Ocampo. Scoping software process lines. *Softw. Process: Improvement Practice*, 14(3):181–197, 2009.

[3] P. Asirelli, M. ter Beek, A. Fantechi, and S. Gnesi. A model-checking tool for families of services. In *Formal Tech. Distr. Sys.*, volume 6722 of *LNCS*, 44–58. 2011.

[4] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: The KobrA approach. In *SPLC: Int. Softw. Prod. Line Conf.*, 289–309, 2000.

[5] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE: Int. Conf. Softw. Eng.*, 702–703, 2004.

[6] A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr, and A. Wise. Little-JIL/Juliette: A process definition language and interpreter. In *ICSE: Int. Conf. Softw. Eng.*, 754–757, 2000.

[7] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *Int. J. Softw. Tools Tech. Transfer*, 14(5):589–612, 2012.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE: Int. Conf. Softw. Eng.*, 335–344, 2010.

[9] P. Clements and L. Northrop. *Software Product Lines–Practices and Patterns*. Addison-Wesley Prof., 2001.

[10] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Int. J. Formal Methods Sys. Design*, 6(1):97–123, 1 1995.

[11] K. Czarnecki and U. Eisenecker. Components and generative programming. In *ESEC/FSE: Euro. Softw. Eng. Conf./Int. Symp. Found. Softw. Eng.*, 2–19, 1999.

[12] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[13] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *Symp. Softw. Reusability*, 109–117, 2001.

[14] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Formal Meth. Open Object-Based Distr. Sys.*, volume 5051 of *LNCS*, 113–131. 2008.

[15] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[16] C. Kästner and S. Apel. Type-checking software product lines-a formal approach. In *ASE: Int. Conf. Auto. Softw. Eng.*, 258–267, 2008.

[17] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE: Int. Conf. Softw. Eng.*, 311–320, 2008.

[18] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, , and S. Apel. FeatureIDE: a tool framework for feature-oriented software development. In *ICSE: Int. Conf. Softw. Eng.*, 611–614, 2009.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP: Euro. Conf. Object-Oriented Prog.*, 220–242, 1997.

[20] S. Knauber. Synergy between component-based and generative approaches. In *ESEC/FSE: Euro. Softw. Eng. Conf./Int. Symp. Found. Softw. Eng.*, 2–19, 1999.

[21] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006.

[22] T. Martínez-Ruiz, F. García, M. Piattini, and F. De Lucas-Consuegra. Process variability management in global software development: a case study. In *ICSSP: Int. Conf. Softw. Sys. Process*, 46–55, 2013.

[23] T. Martínez-Ruiz, J. Münch, F. García, and M. Piattini. Requirements and constructors for tailoring software processes: a systematic literature review. *J. Softw. Quality*, 20(1):229–260, 2012.

[24] S. Mohalik, S. Ramesh, J.-V. Millo, S. N. Krishna, and G. K. Narwane. Tracing spls precisely and efficiently. In *SPLC: Int. Softw. Prod. Line Conf.*, 186–195, 2012.

[25] K. Pohl and A. Metzger. Variability management in software product line engineering. In *ICSE: Int. Conf. Softw. Eng.*, 1049–1050, 2006.

[26] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *IEEE Int. Conf. Requir. Eng.*, 139–148, 2006.

[27] B. Simidchieva and L. Osterweil. Characterizing process variation: NIER track. In *ICSE: Int. Conf. Softw. Eng.*, 836–839, 2011.

[28] B. I. Simidchieva, S. Engle, M. Clifford, A. C. Jones, S. Peisert, M. Bishop, L. A. Clarke, and L. J. Osterweil. Modeling and analyzing faults to improve election process robustness. In *EVT/WOTE: Electronic Voting Tech. Workshop/Workshop Trustworthy Elections*, 2010.

[29] B. I. Simidchieva and L. J. Osterweil. Categorizing and modeling variation in families of systems: a position paper. In *ESCA: Euro. Conf. Softw. Arch.*, 316–323, 2010.

[30] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: a framework for modeling variability in software product families. In *Software Product Lines*, volume 3154 of *LNCS*, 25–27, 2004.

[31] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. In *ICSE: Int. Conf. Softw. Eng.*, 11–21, 2002.

[32] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Int. Conf. Generative Prog. Comp. Eng.*, 95–104, 2007.

[33] H. Washizaki. Building Software Process Line Architectures from Bottom Up. In *Product-Focused Softw. Process Improvement (PROFES)*, 415–421, 2006.

[34] D. M. Weiss and C. T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley, 1999.

[35] A. Wise. Little-JIL 1.5 Language Report. Tech. report, Computer Science, University of Massachusetts, Amherst, MA, 2006.